

こわくない



git

Presented by Kota Saito @ksaito

こんな人が対象者

一応ブランチ作ってコミットしてるけど
実は、マージとかよくわかってない……

エラーが出た時に対処できない……

rebase しちゃダメって何でなの？



Git よくわかんない！

怖い!!!

＼怖くないよ／



git

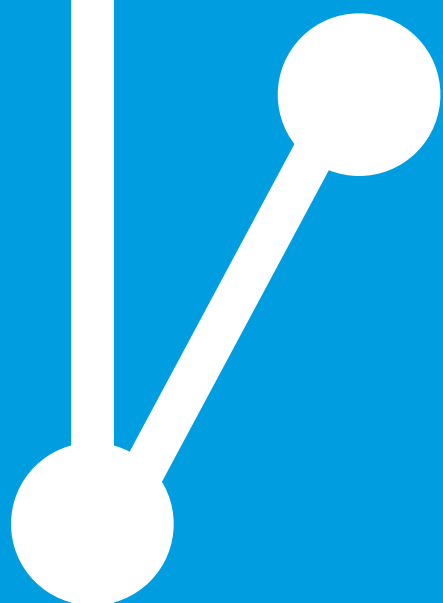


#1 コミットとブランチ

#2 二種類のマージ

#3 リベースの功罪

コミットとブランチ



#1

そもそも、コミットって
なんだっけ……？



コミットに入ってる情報

リビジョン (SHA-1 ハッシュ)

例: 23cdd334e6e251336ca7dd34e0f6e3ea08b5d0db

Author (コミットを作成した人)

例: オープンソースプロジェクトにパッチを送った人

Committer (コミットを適用した人)

例: 受け取ったパッチを取り込んだ人

ファイルのスナップショット (tree)

コミットで変更されたファイルを含むツリー (説明は省略)

1つ前のコミットのリビジョン

例: 4717e3cf182610e9e82940ac45abb0d422a76d77

コミットに入ってる情報

リビジョン (SHA-1 ハッシュ)

例: 23cdd334e6e251336ca7dd34e0f6e3ea08b5d0db

Author (コミットを作成した人)

例: オープンソースプロジェクトにパッチを送った人

Committer (パッチを受け取った人)

例: 受け取ったパッチをコミットした人

ファイルのスナップショット (tree)

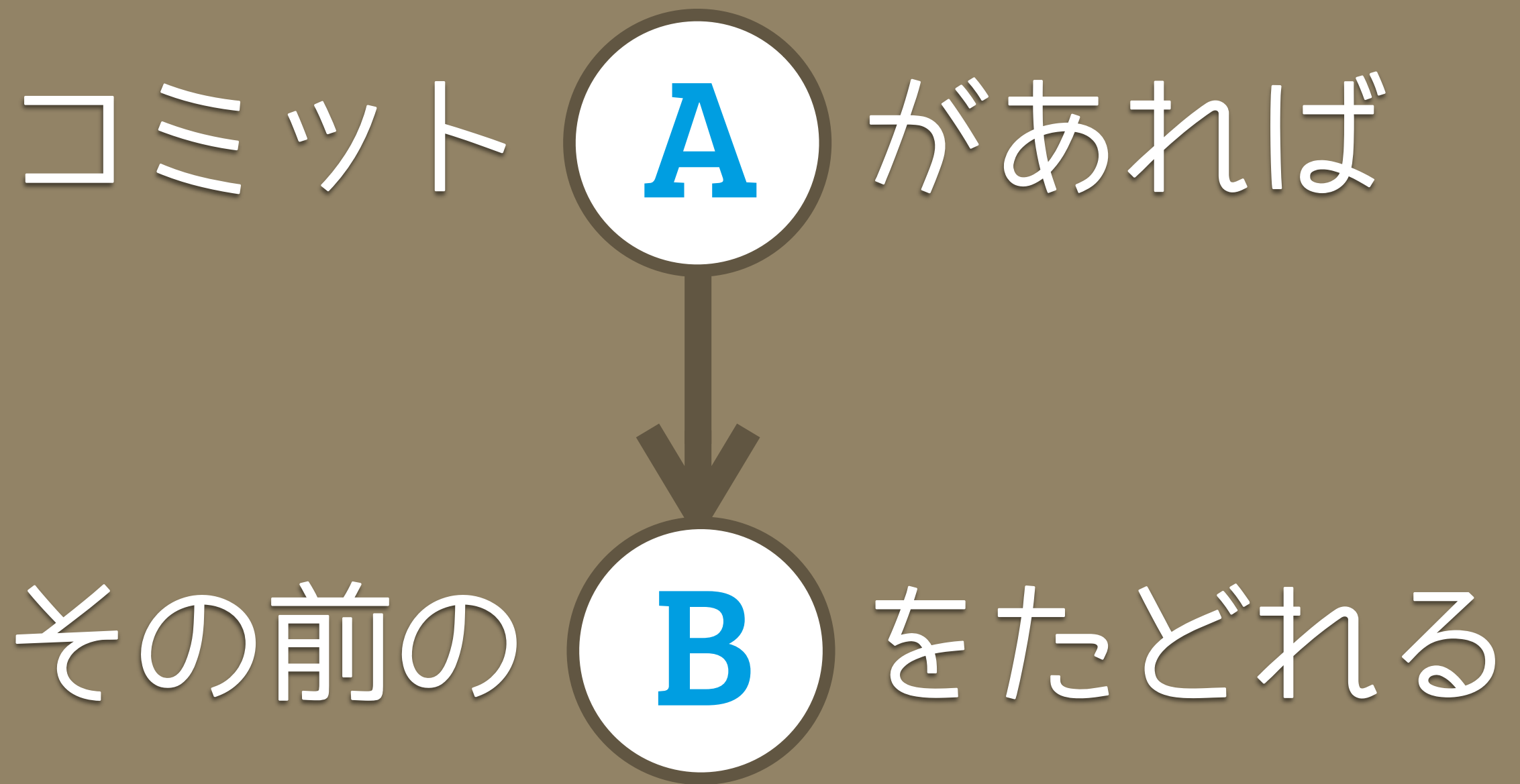
コミットで変更されたファイルを含むツリー (説明は省略)

1つ前のコミットのリビジョン

例: 4717e3cf182610e9e82940ac45abb0d422a76d77

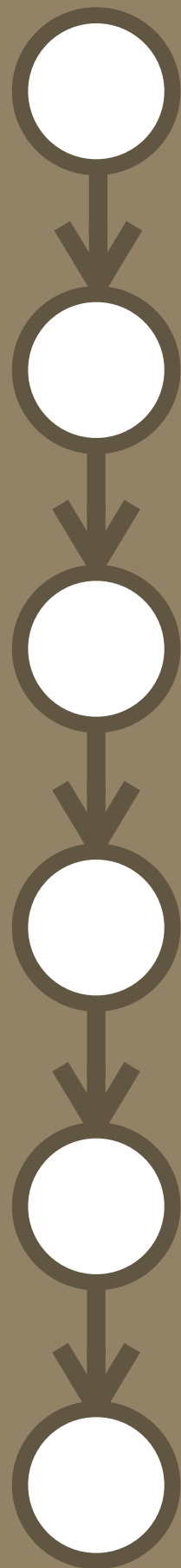
これ重要!!

コミット  があれば







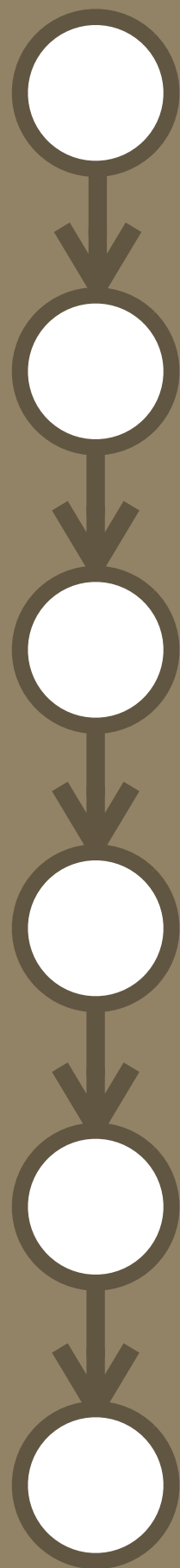


最新のコミット



たどれる!!

最古のコミット



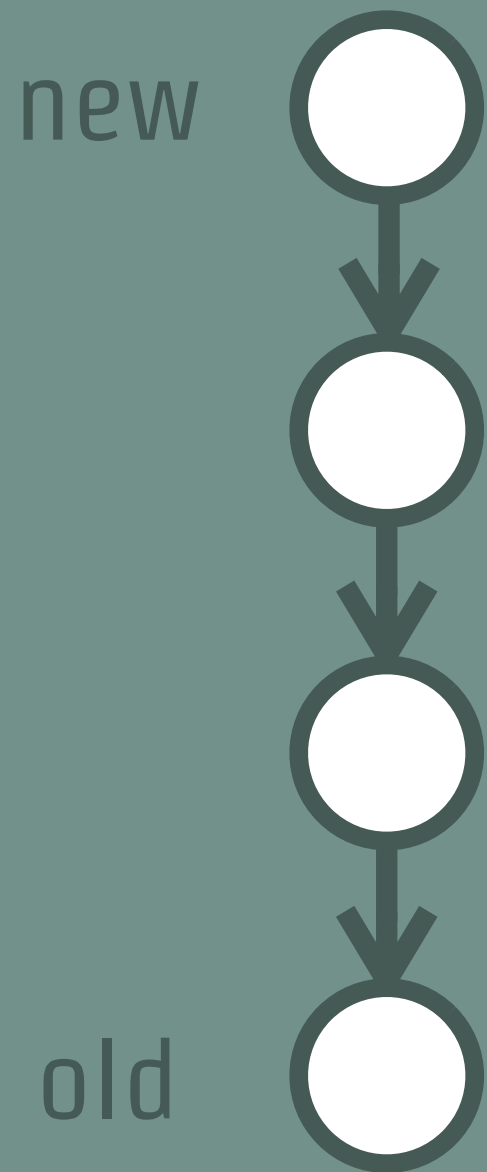
この概念 =

コミットグラフ

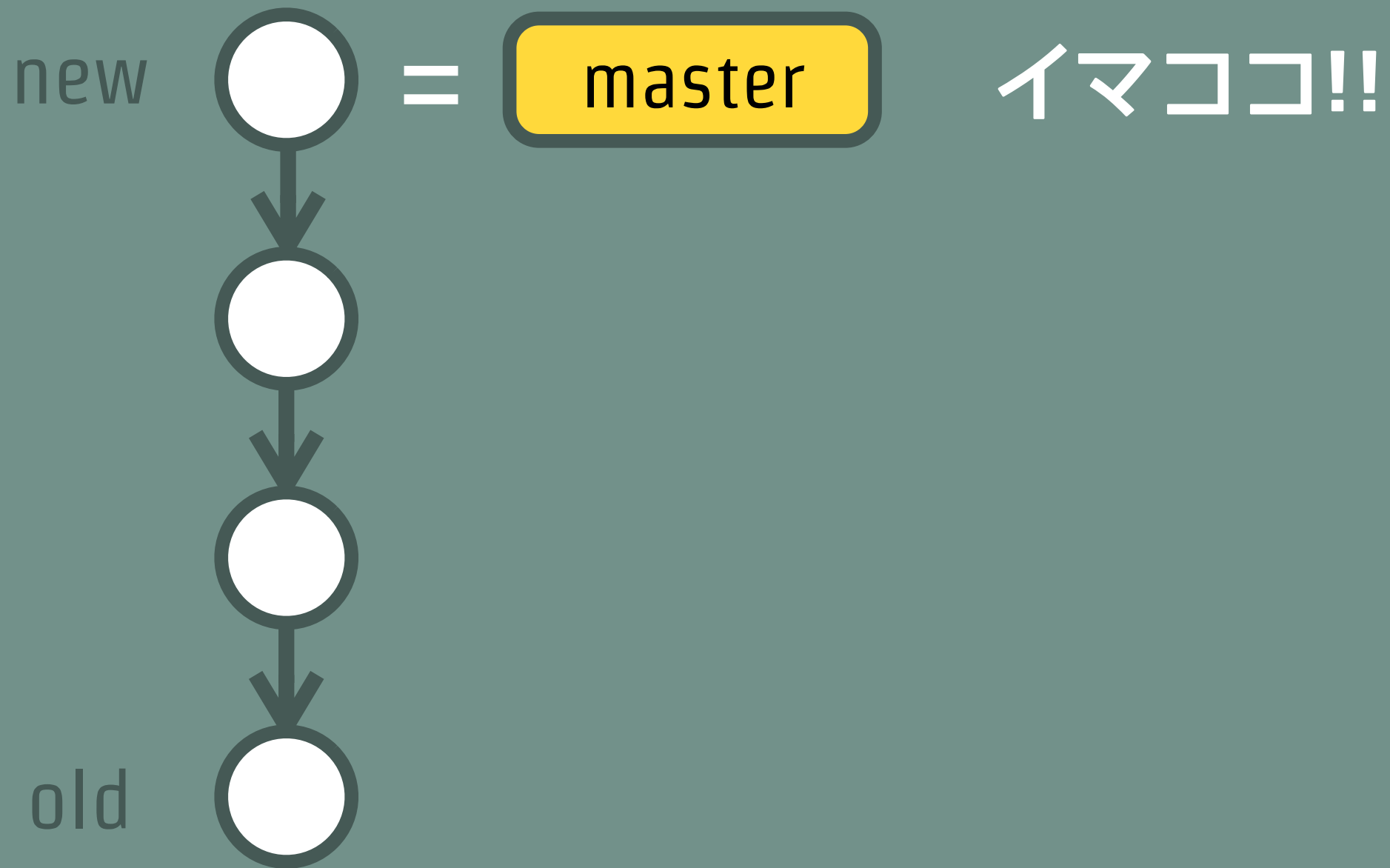
＼テストに出るぞ／

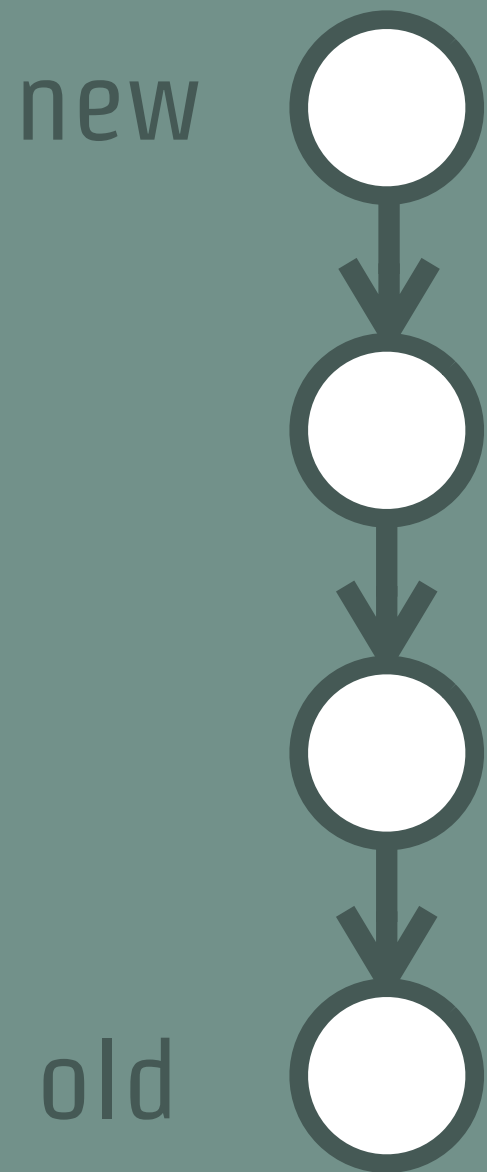
で、ブランチって
なんだっけ……？





例えば **master** に
コミットが4回されたあとの
こんなコミットグラフ





さらに **master** にもう1回
コミットがされると...

new!



=



イマココ!!



old

つまり



ブランチは最新コミットの別名!!

ブランチ名の代わりに
リビジョンを指定したり

```
git checkout -b foo master
```

||

```
git checkout -b foo 4717e3cf1826
```


リビジョンの代わりに
ブランチ名を指定したり

```
git show 4717e3cf1826
```

||

```
git show master
```

できる!!



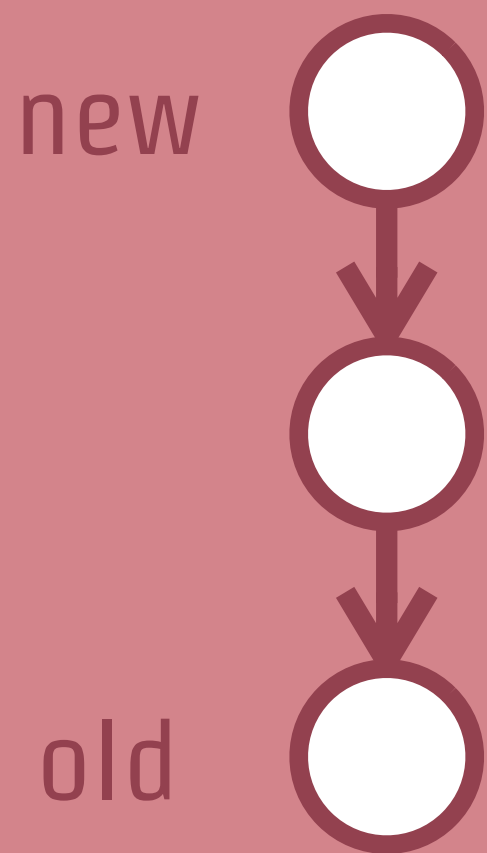
あれ？ 枝分かれする
ブランチの話は？

てか、masterって
ブランチだったのか

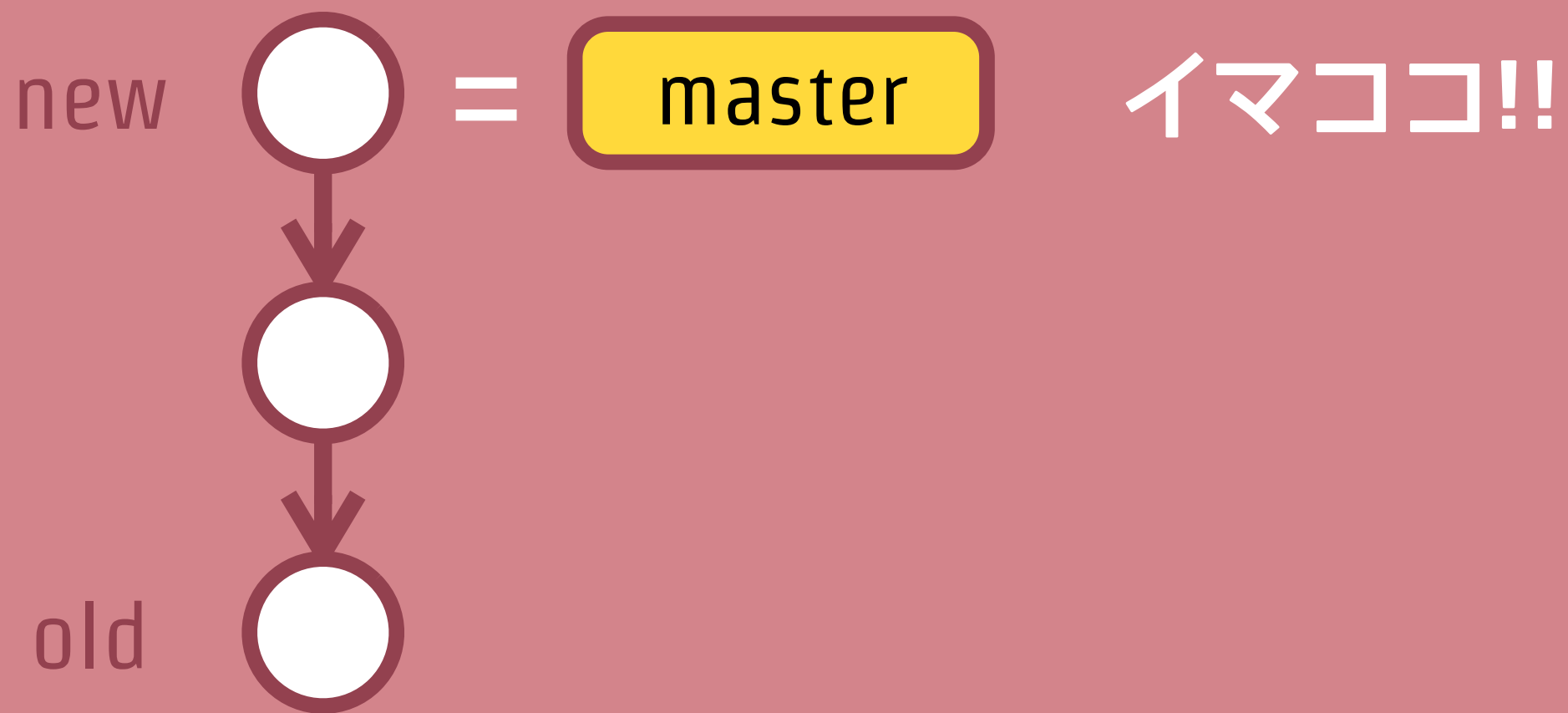
git branch topic master

git branch topic master

master から topic を作る

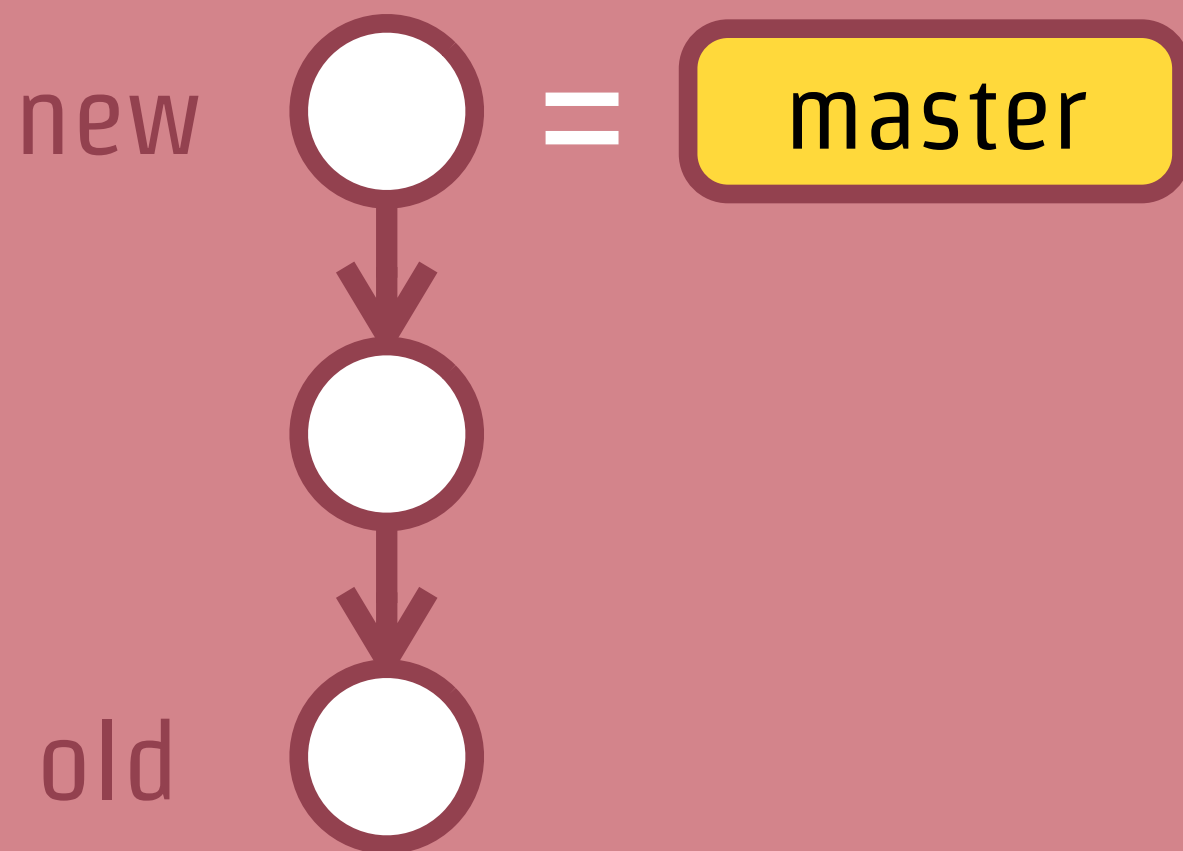


例えば **master** に
コミットが3回されたあとの
こんなコミットグラフ



git branch topic master

してみると.....




!?



つまり

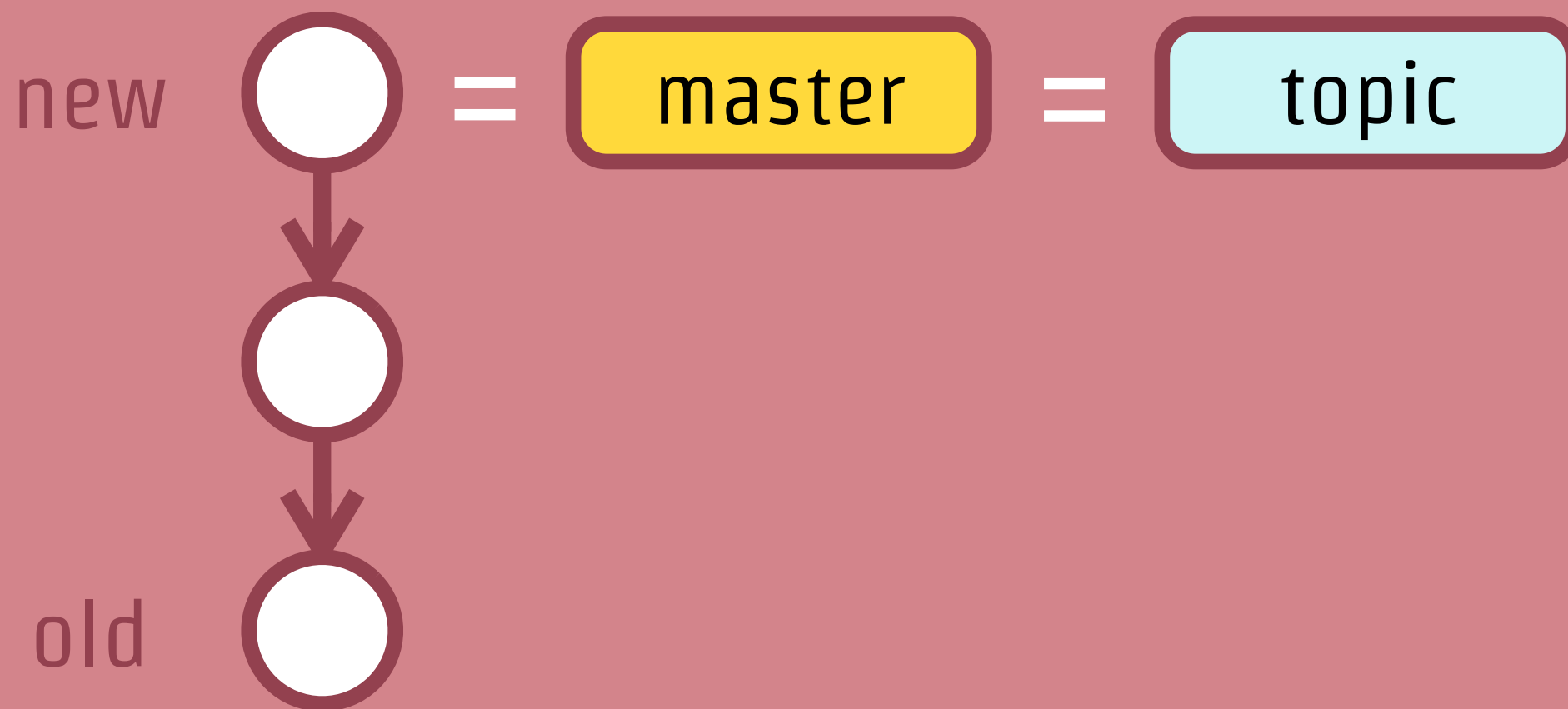


ブランチを作る =
コミットのさらなる別名を作る



枝分かれしてなくね？
スルーですか？

この状態から **topic** に
コミットしてみると……



new!



=



伸びた!!



=



old

master

にコミットしてみると...

new!



=

topic

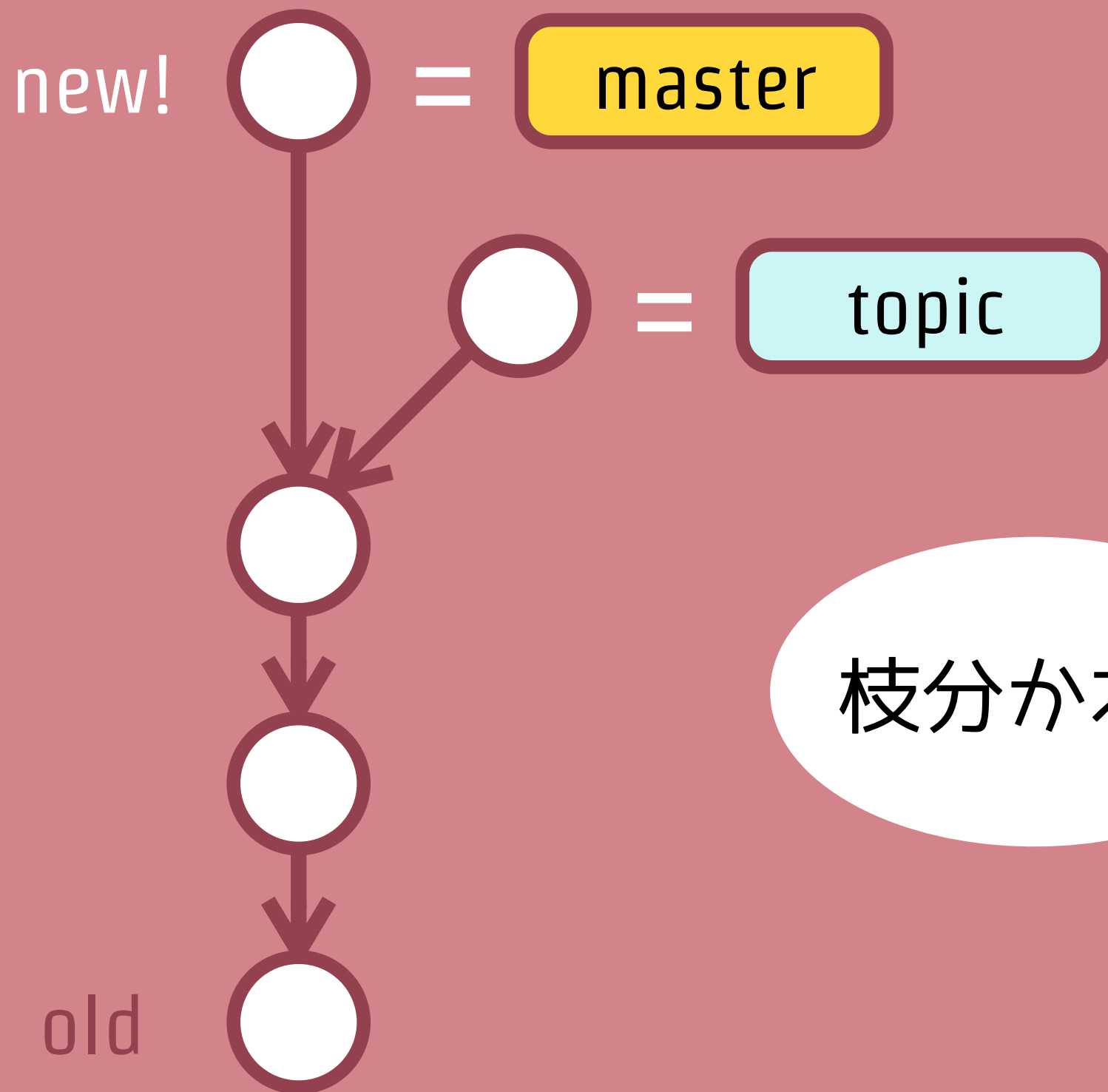


=

master



old



枝分かれ!!



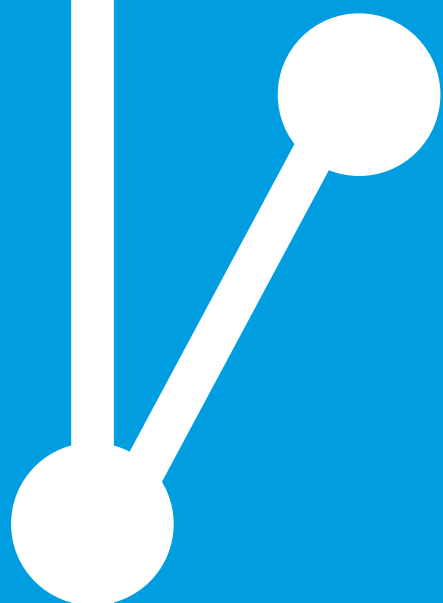
Point

初めてコミットした時点で
枝分かれます。

コミットとブランチ

- ・ コミットには「1つ前のコミット」が含まれるので、最古までたどっていける
→ コミットグラフ
- ・ ブランチは、そのブランチでの最新のコミットの別名に過ぎない
- ・ 枝分かれば、それぞれのブランチでコミットされて初めて発生する

二種類^のマージ



#2

Question

gitのマージには

2種類ある事

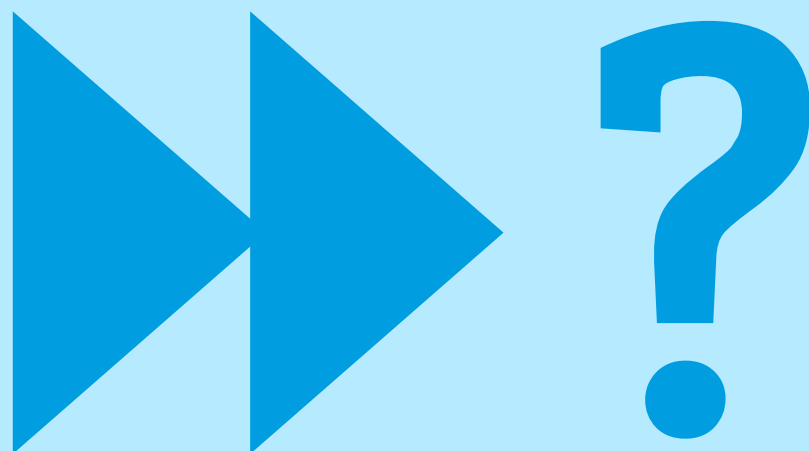
知ってましたか？

Fast-Forward

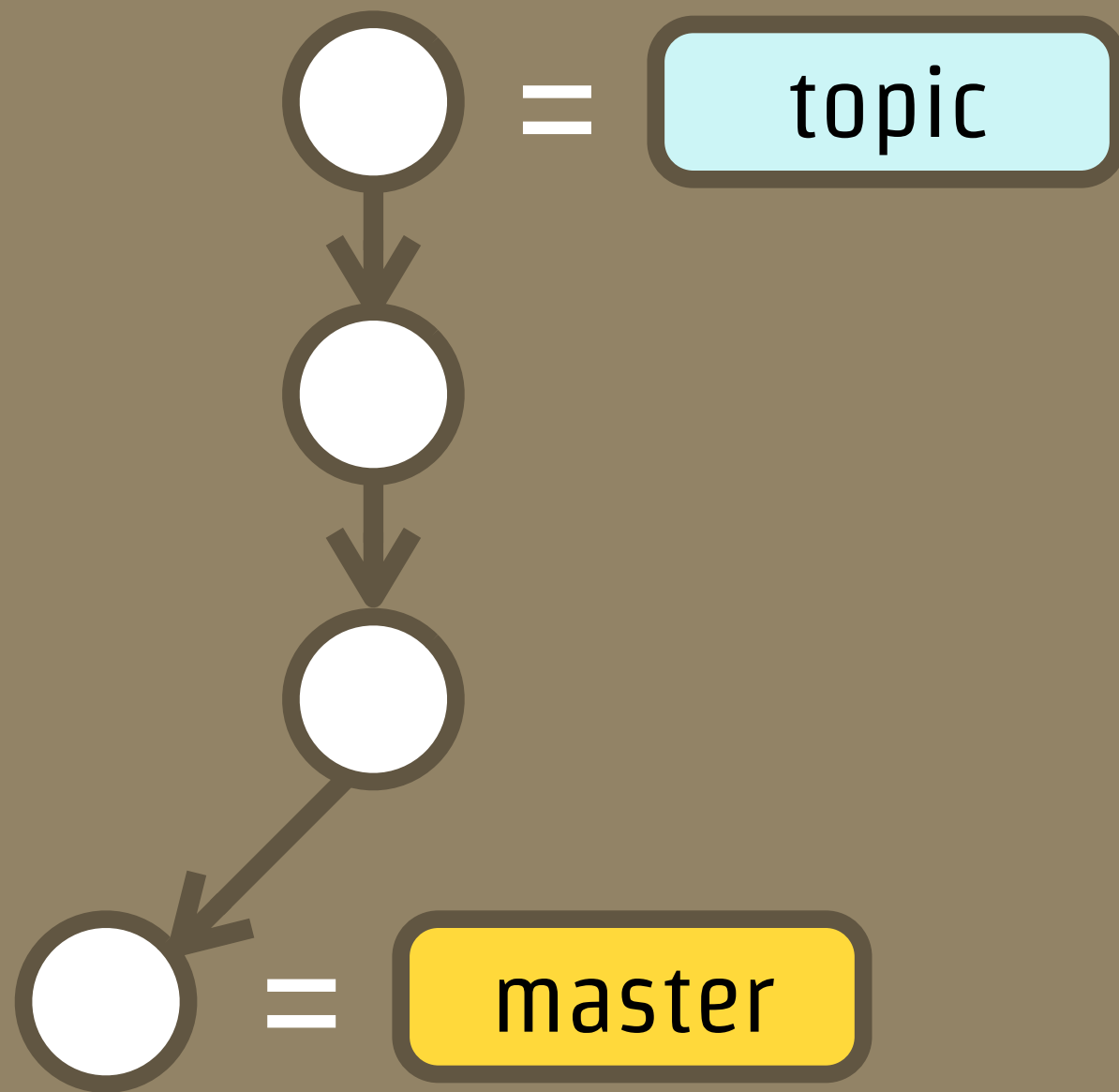
and

Non Fast-Forward

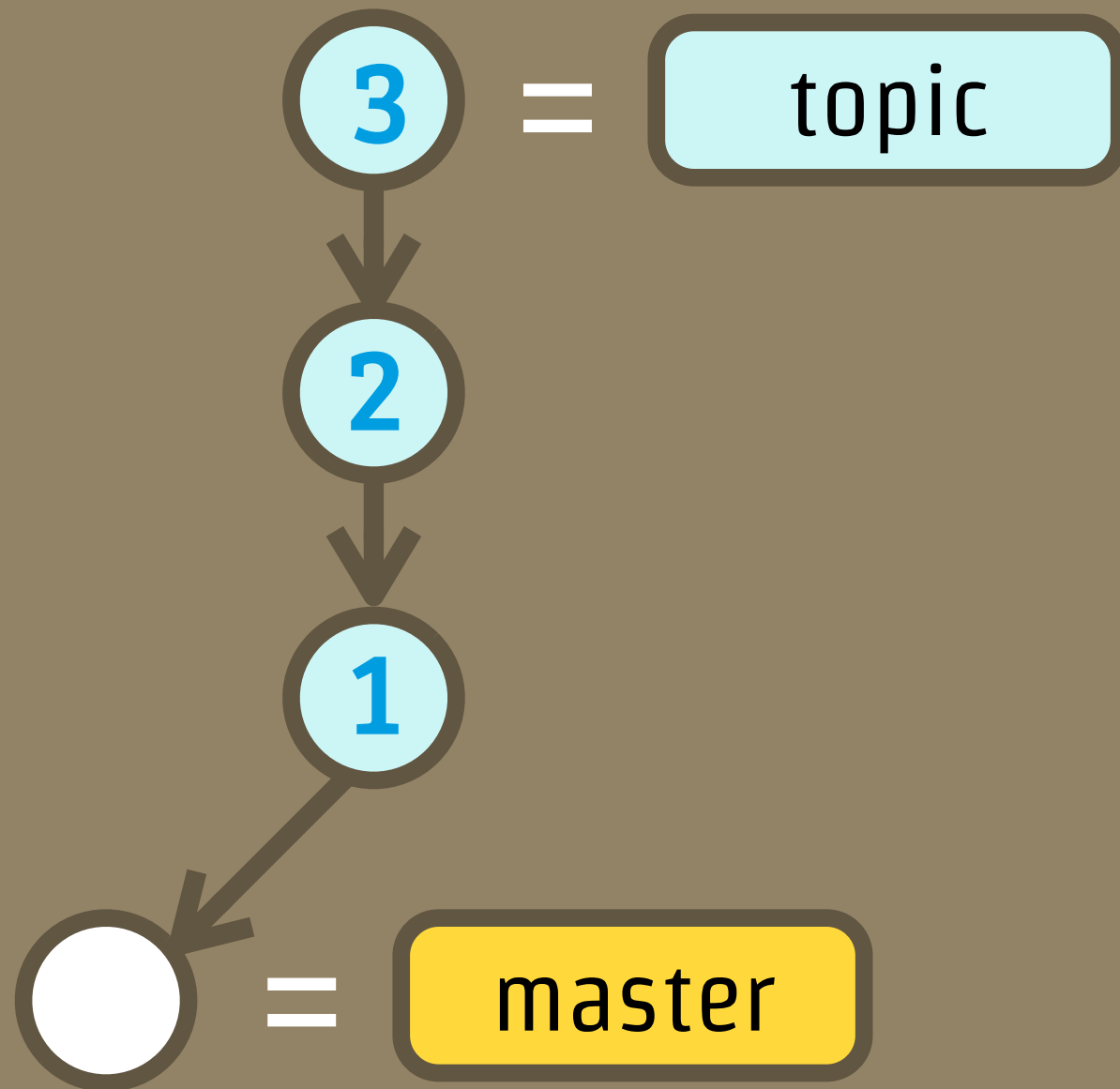
Fast-Forward = 早送り



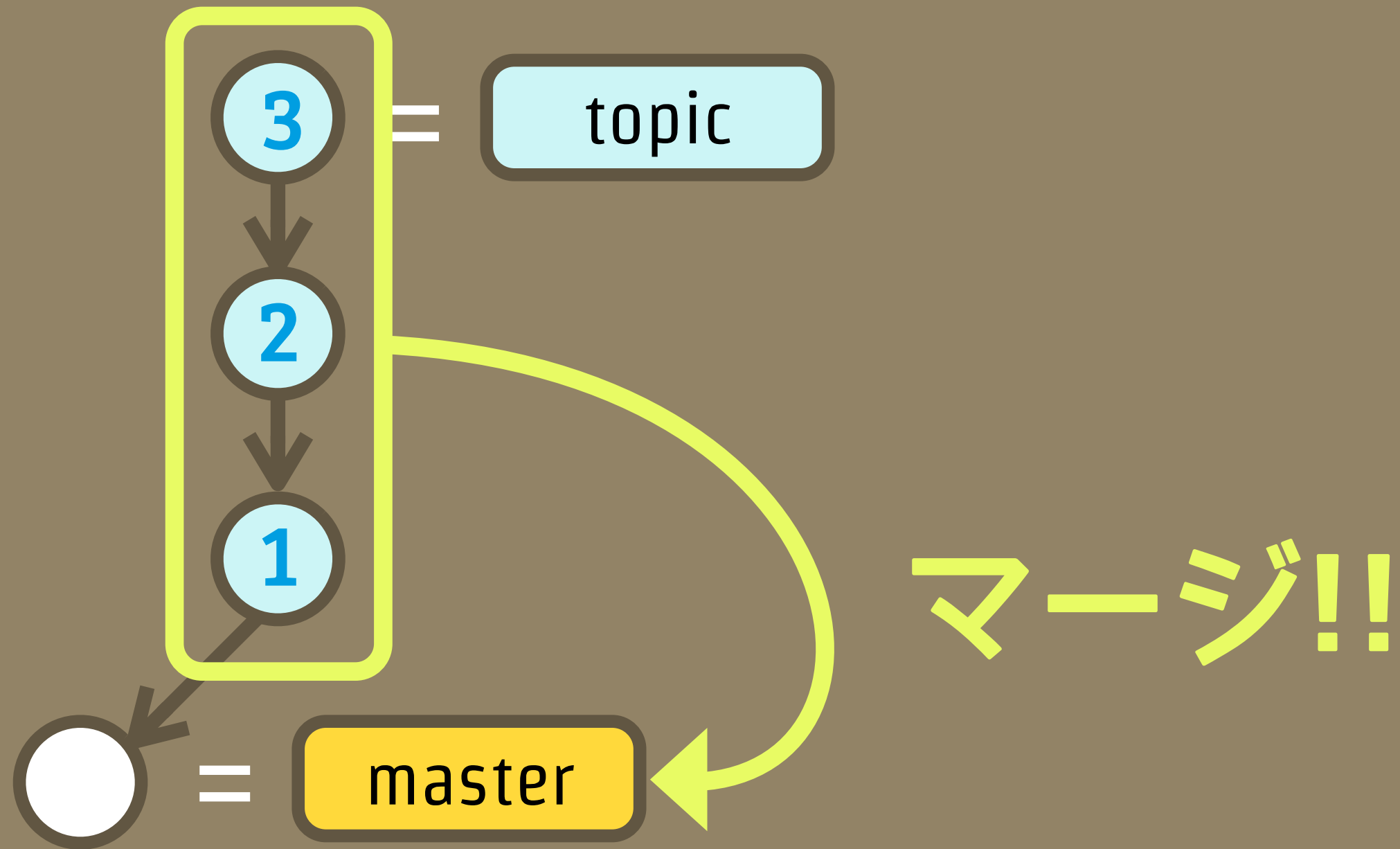
例えばこんなコミットグラフ



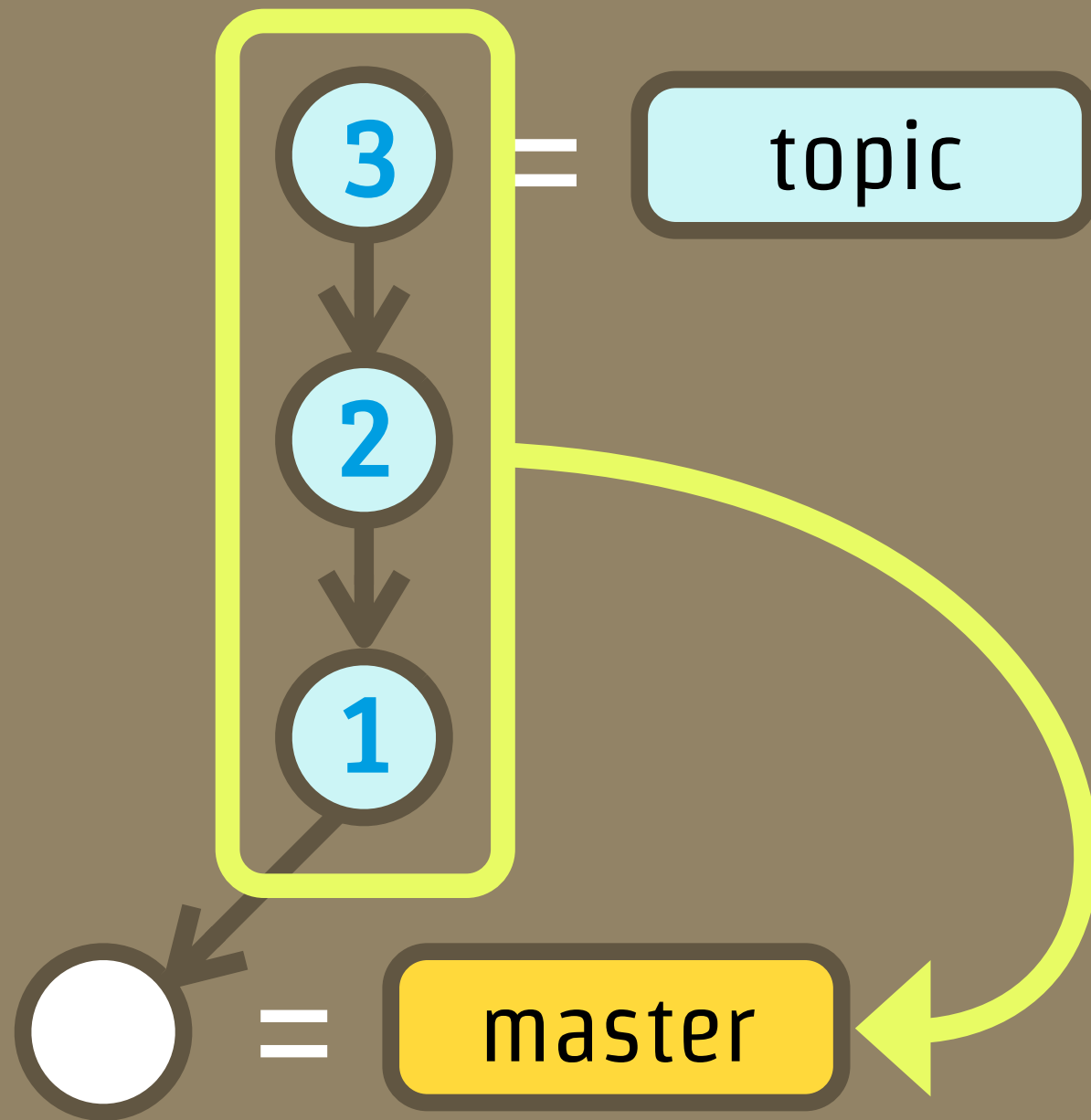
topic ブランチに
3回コミットした



git merge topic

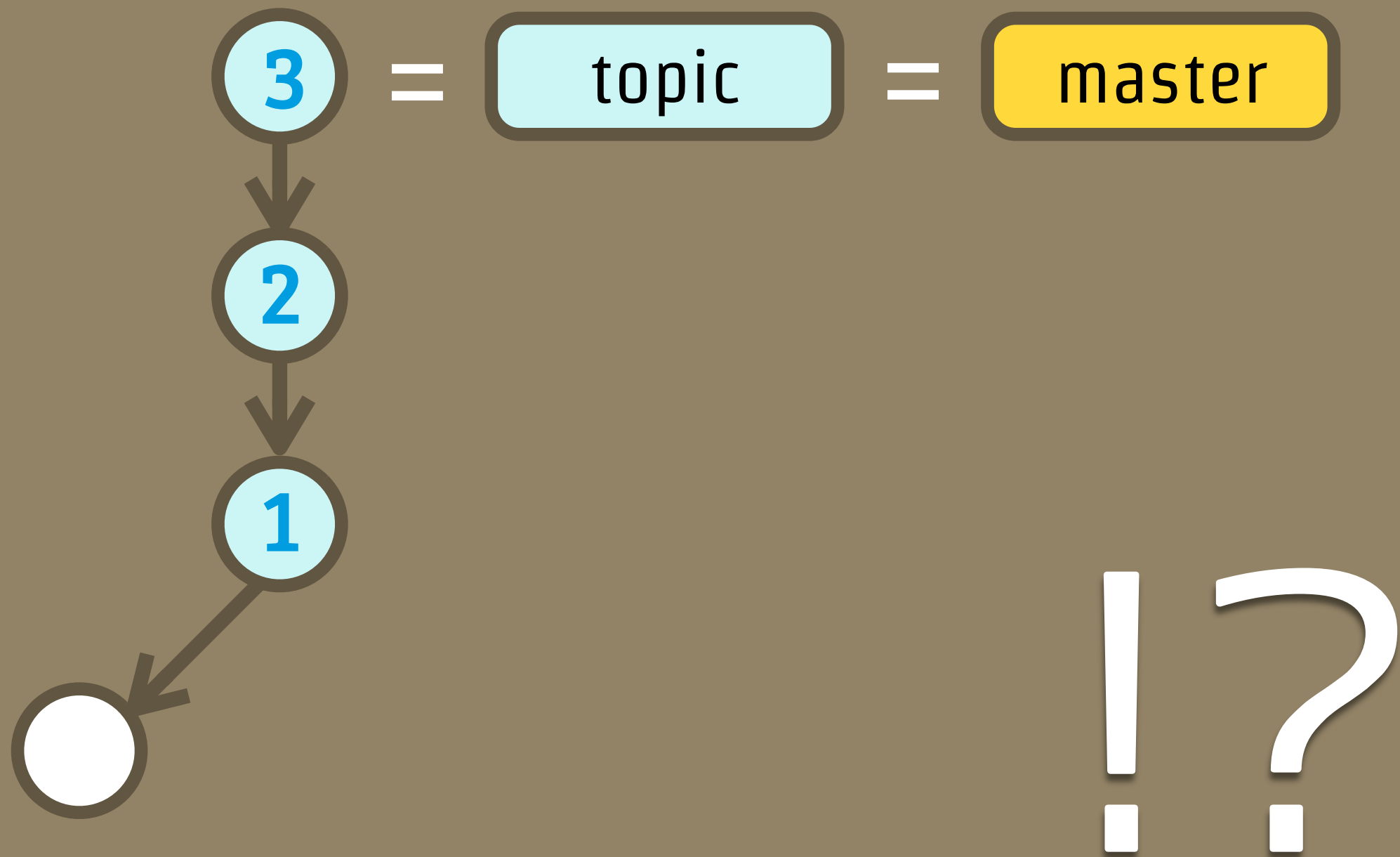


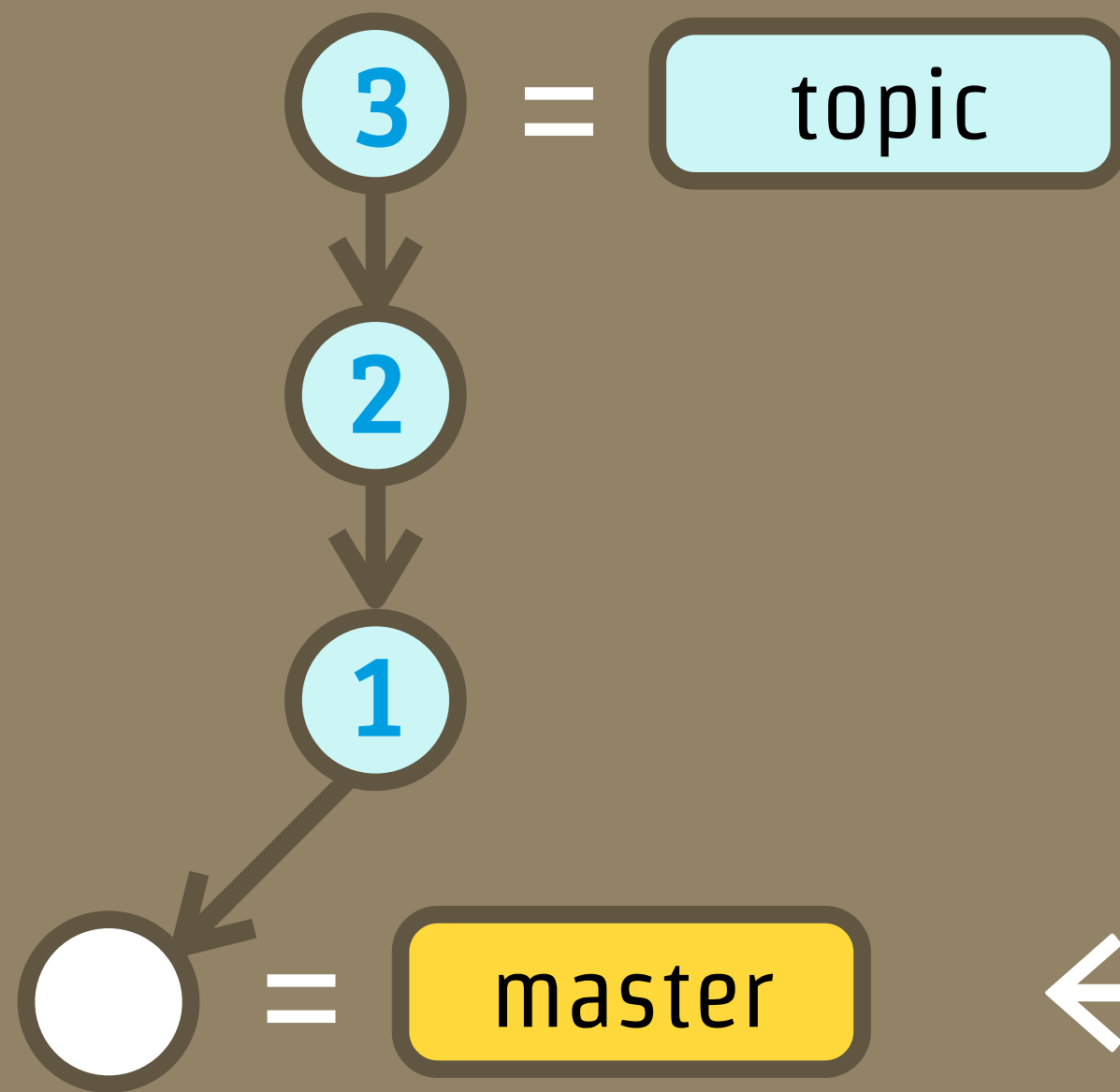
git merge topic



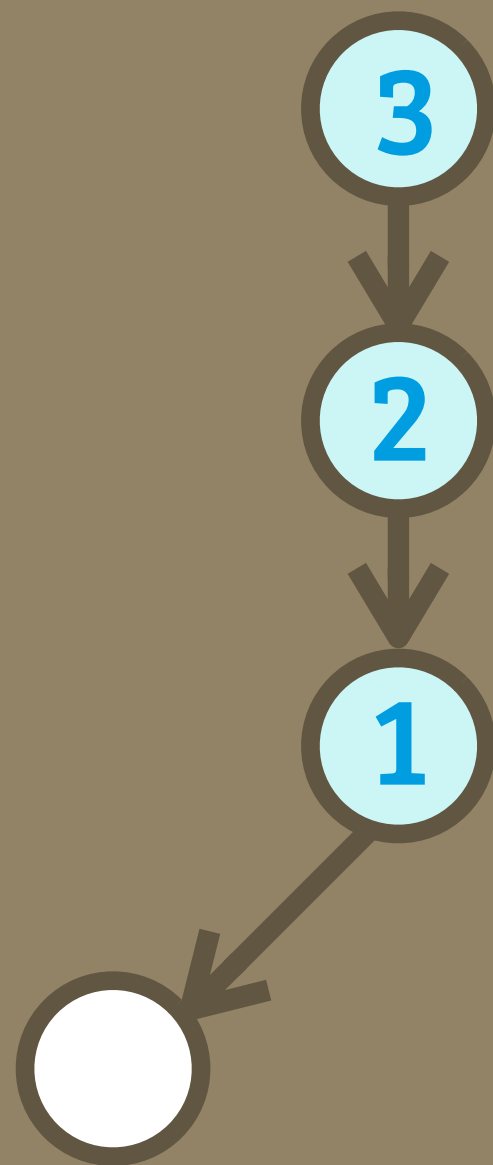
マージ!!

グラフはどうなる？





← これが



=

topic

=

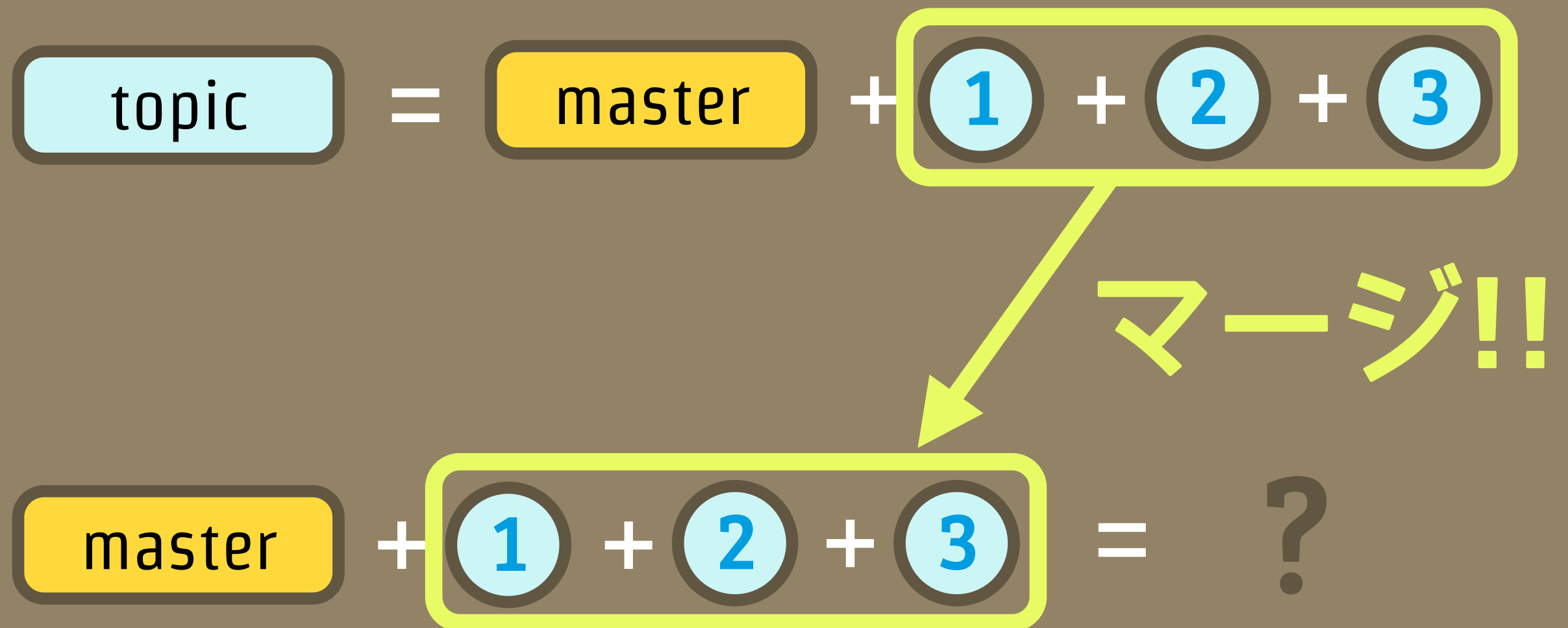
master

ここに↑
移動しただけ

Why?

$$\text{topic} = \text{master} + 1 + 2 + 3$$

master



$$\text{topic} = \text{master} + 1 + 2 + 3$$

$$\text{master} + 1 + 2 + 3 = \text{topic}$$

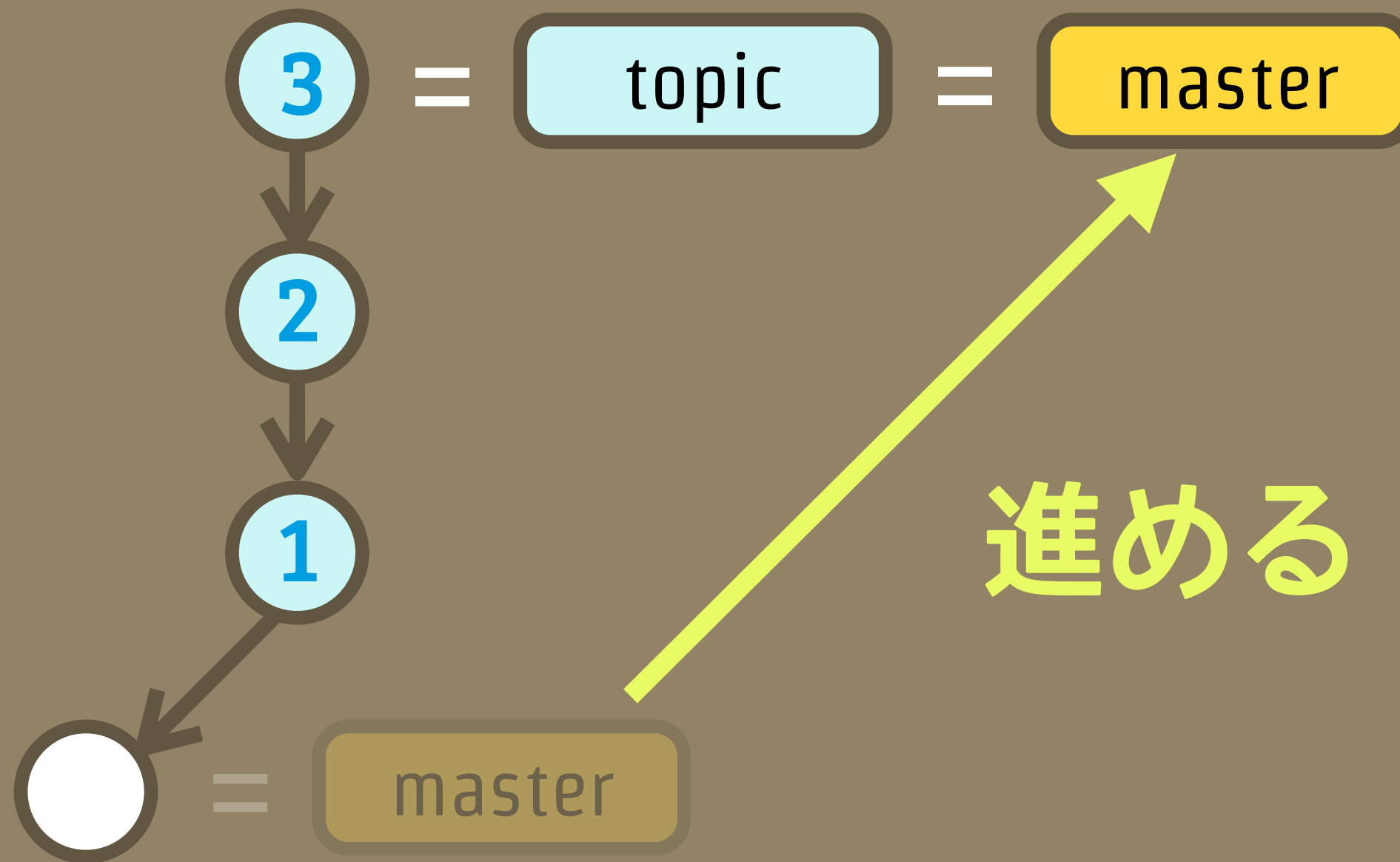
$$\text{topic} = \text{master} + ① + ② + ③$$

マージ後の

$$\text{master} = \text{topic}$$

中の人「どうせ同じ内容になるなら
いちいちマージしなくてよくね？」

中の人「マージ後に topic と同じ内容になるなら
topic のところまで進めちゃおうぜww」



これが **Fast-Forward**

Fast-Forward

マージのかわりに早送り。

＼ト"ヤア...／



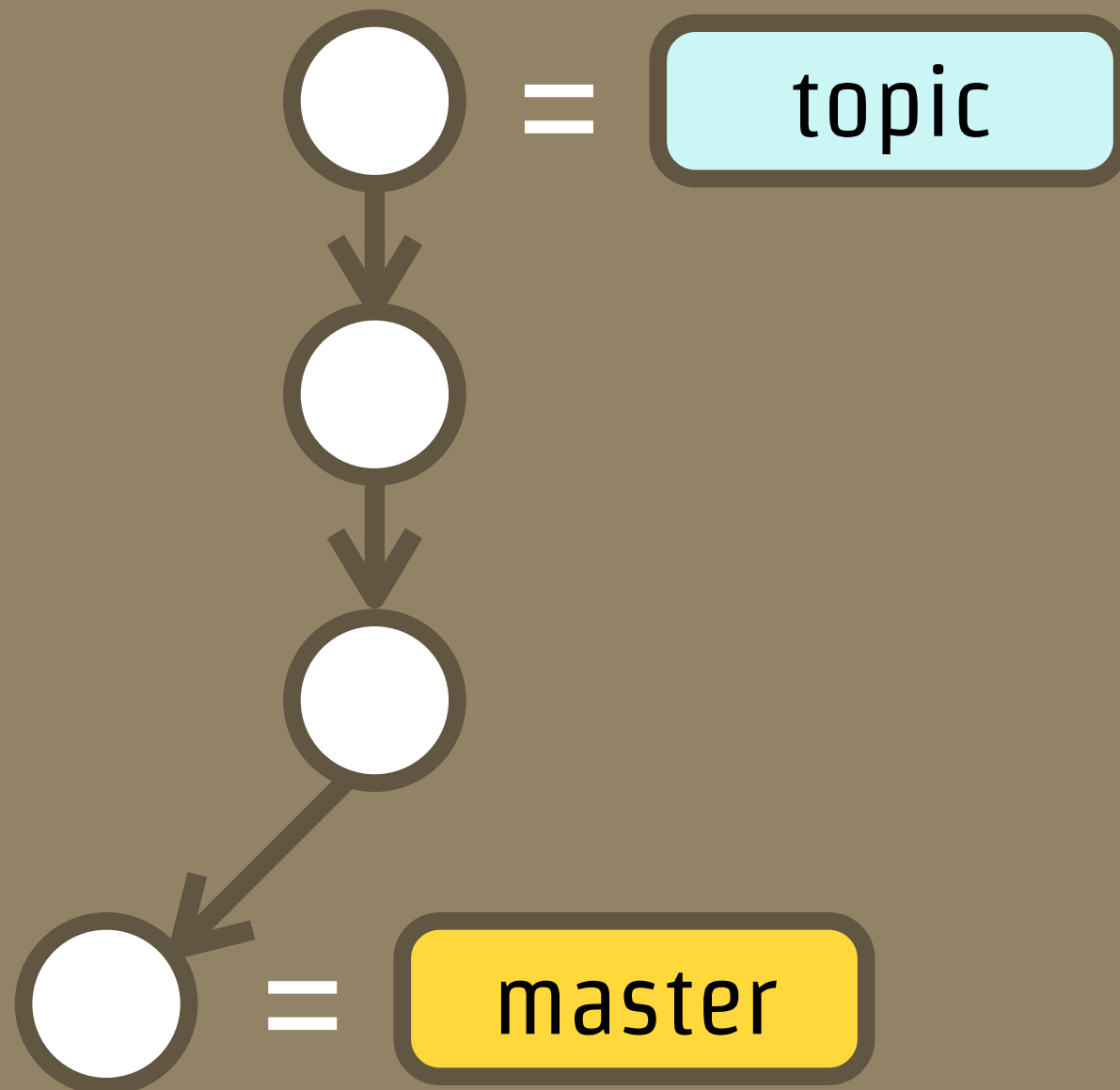
git

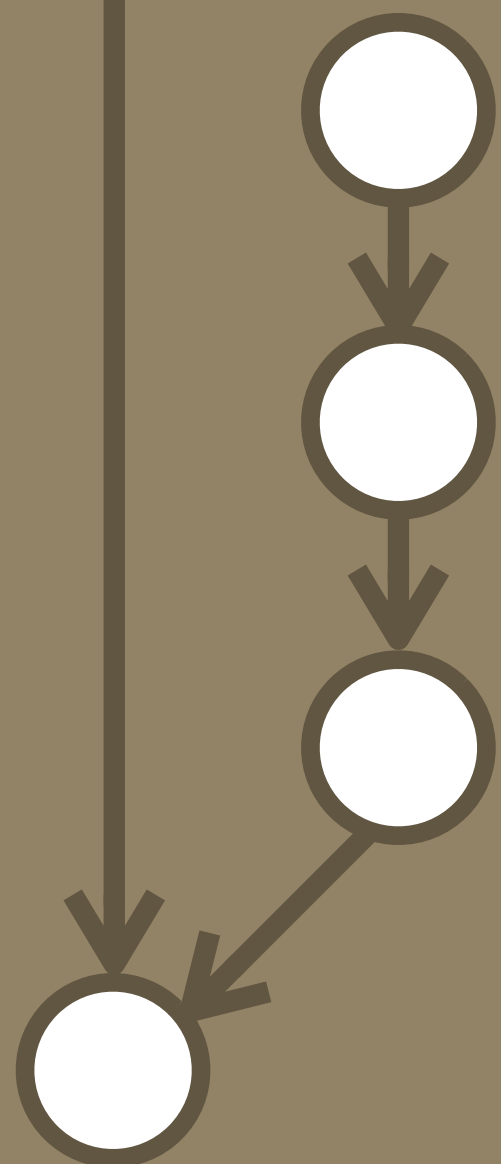
Non Fast-Forward

さっきのコミットグラフから

master

にコミットすると...





枝分かれ!!



マージ後の

master

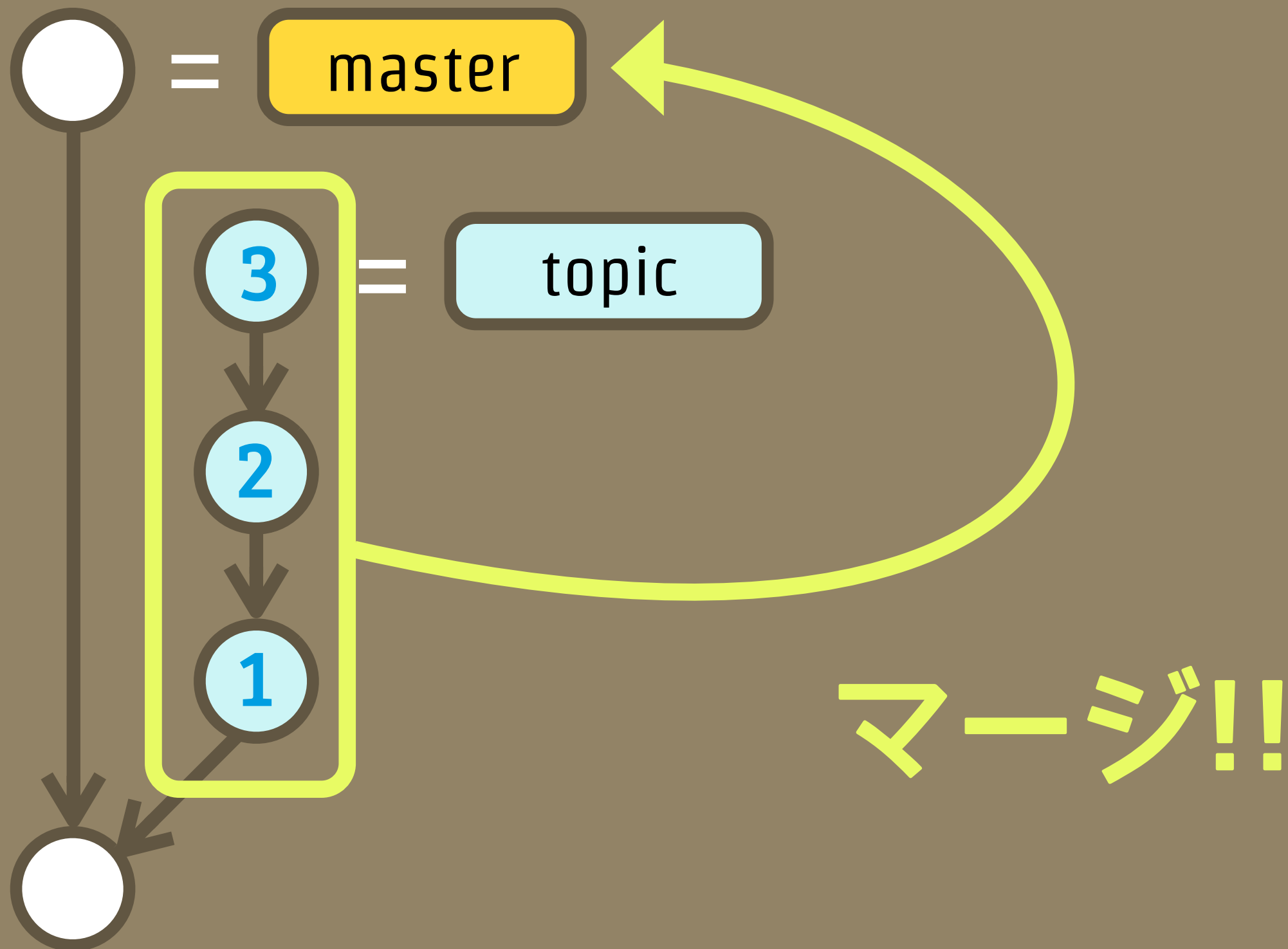
≠

topic

中の人「早送りできない><」

中の人「ちゃんとマージするかー」

git merge topic



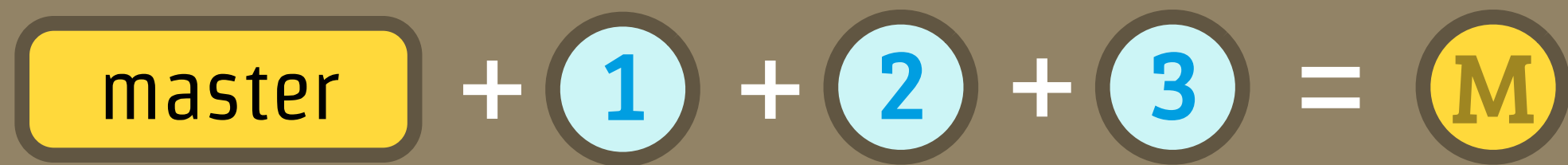
～中の人計算中～

「えーと master の内容と
topic のコミットを比較して…」

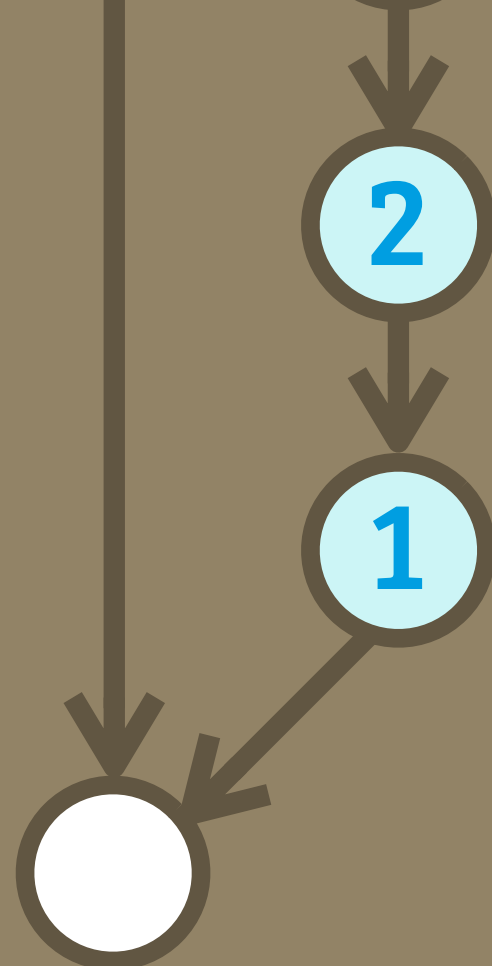
「衝突してたら教えてあげなきゃ…」



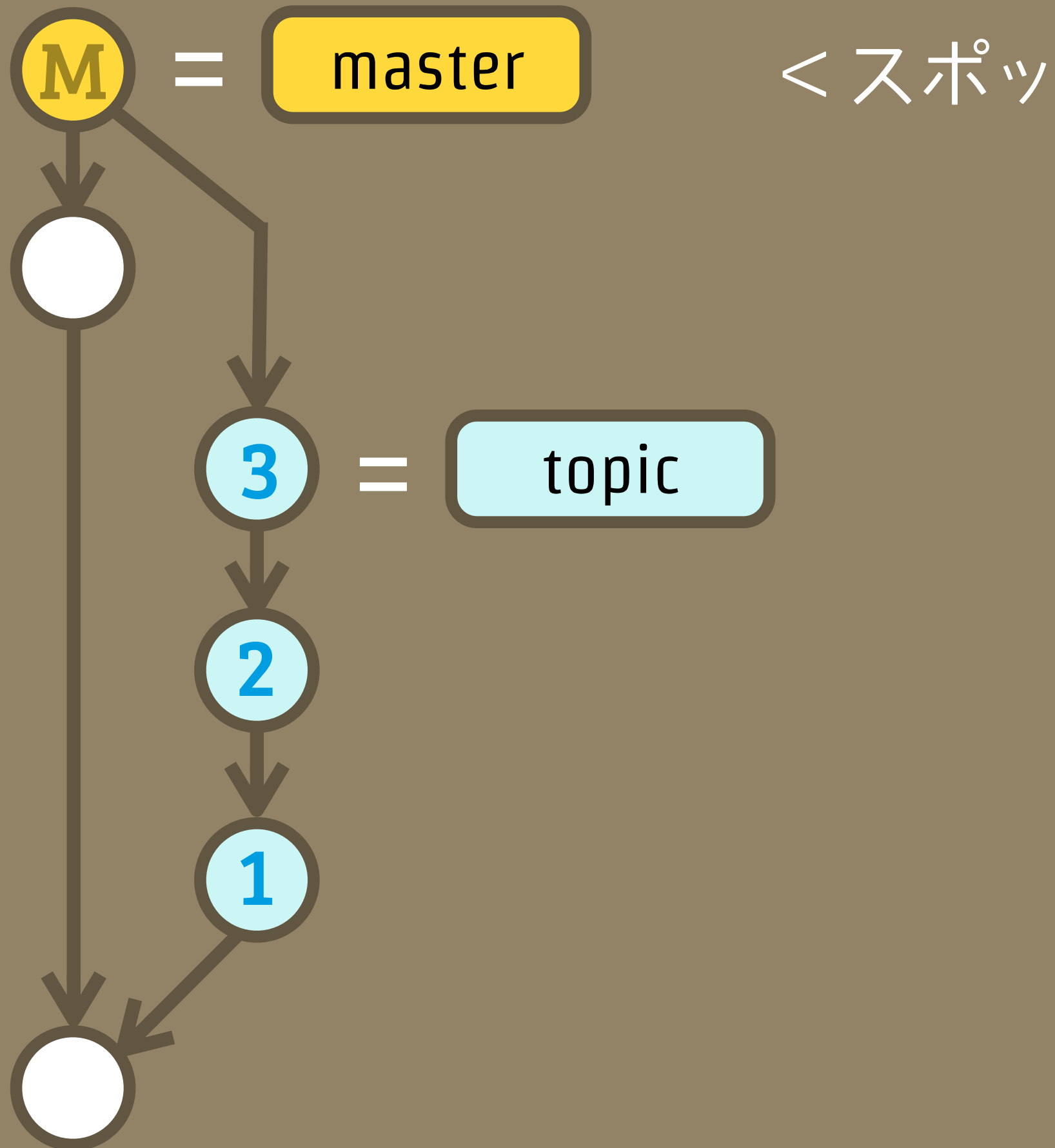
中の人「全部まぜちゃえー」

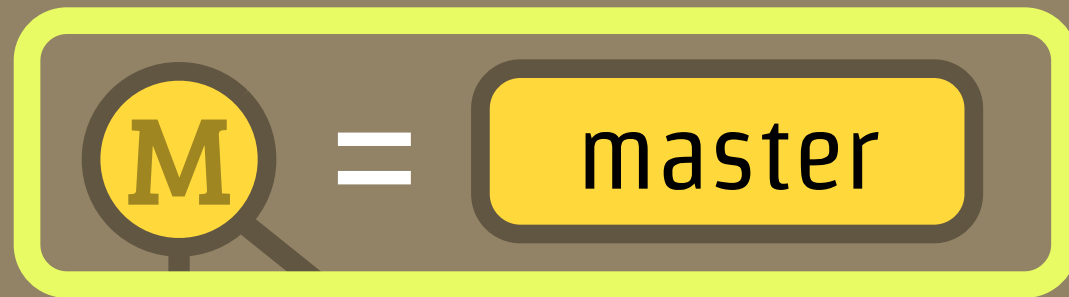


中の人「マージ結果ができたよ！」

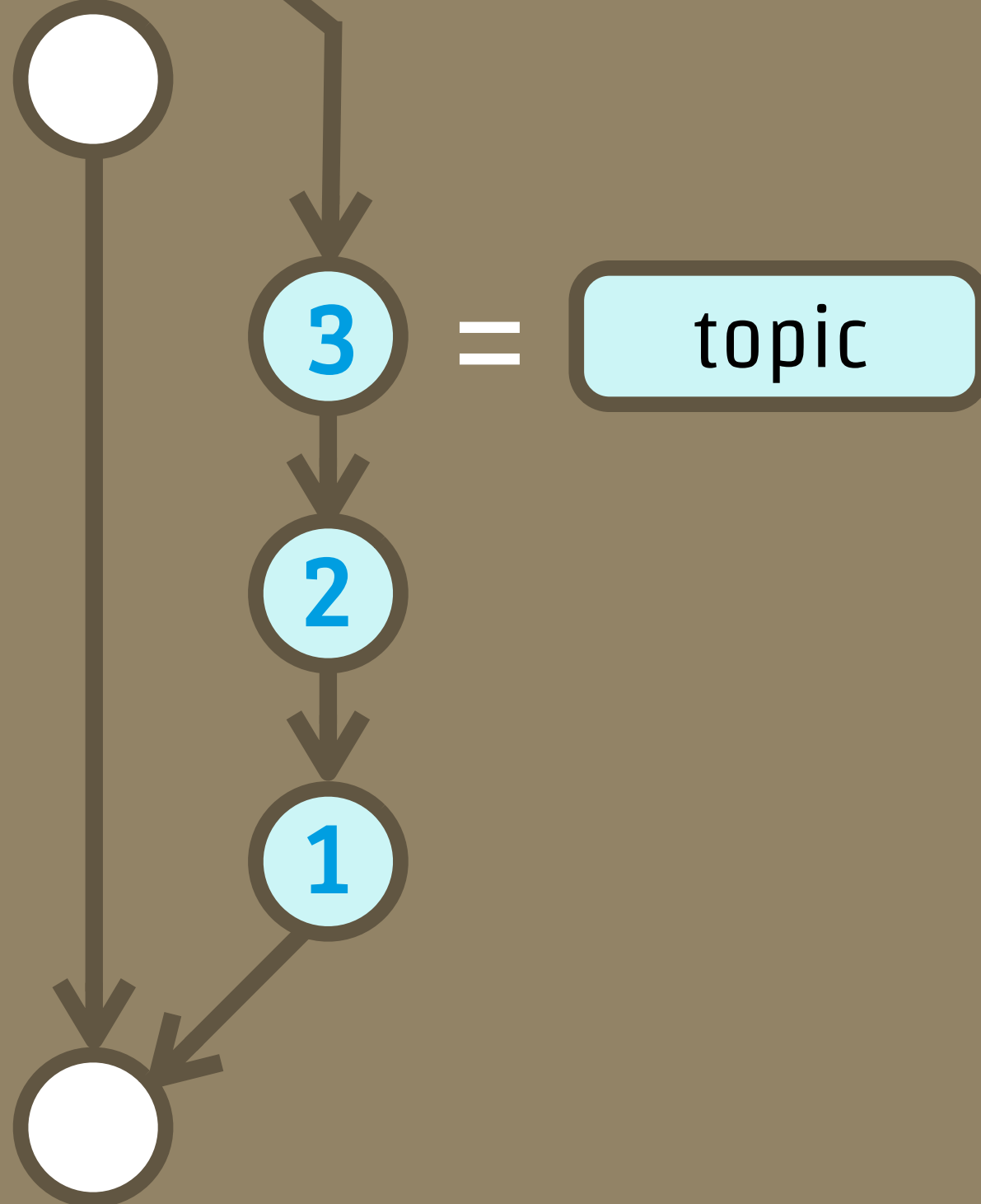


中の人「そおい!!!」





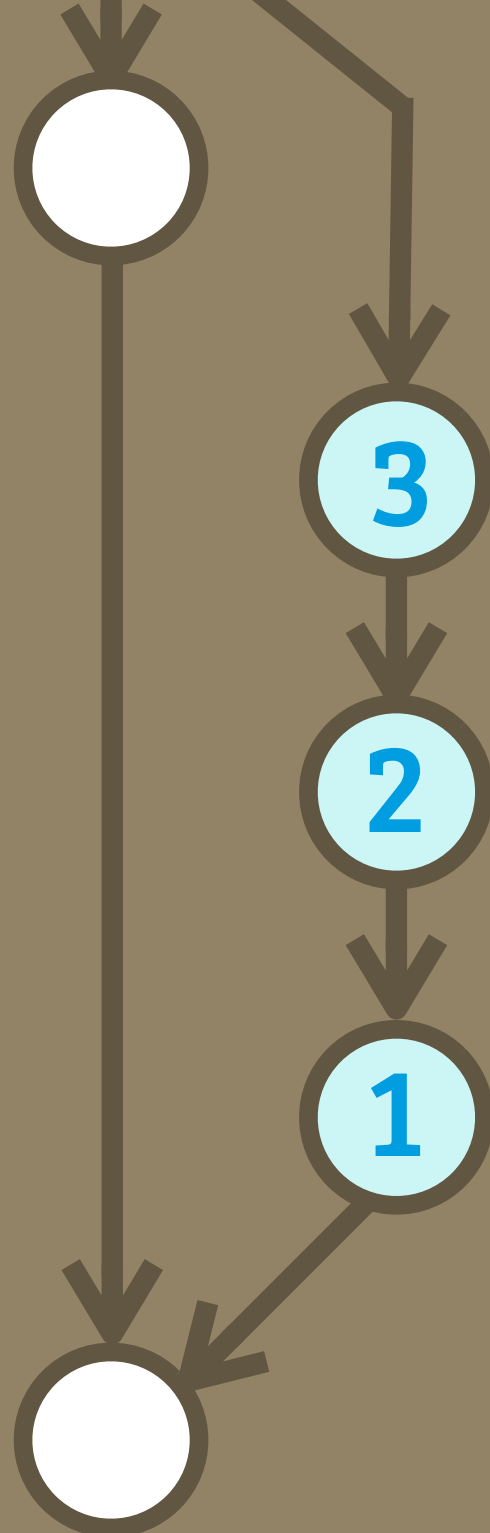
← マージの結果
作られたコミット





← マージの結果
作られたコミット

マージコミット



＼テストに出るぞ／

Non Fast-Forward

Non Fast-Forward

早送りじゃないマージ。

＼ト”ヤヤヤア…／



git

ちなみに

git merge topic

Fast-Forward

Non Fast-Forward

早送りできればする、無理なら普通のマージ

git merge --no-ff topic

Non Fast-Forward

常に普通のマージ

git merge --ff-only topic

Fast-Forward

常に早送りする（できなければエラーとする）

ところが

議論がある

常に `git merge --no-ff` すべきだ！

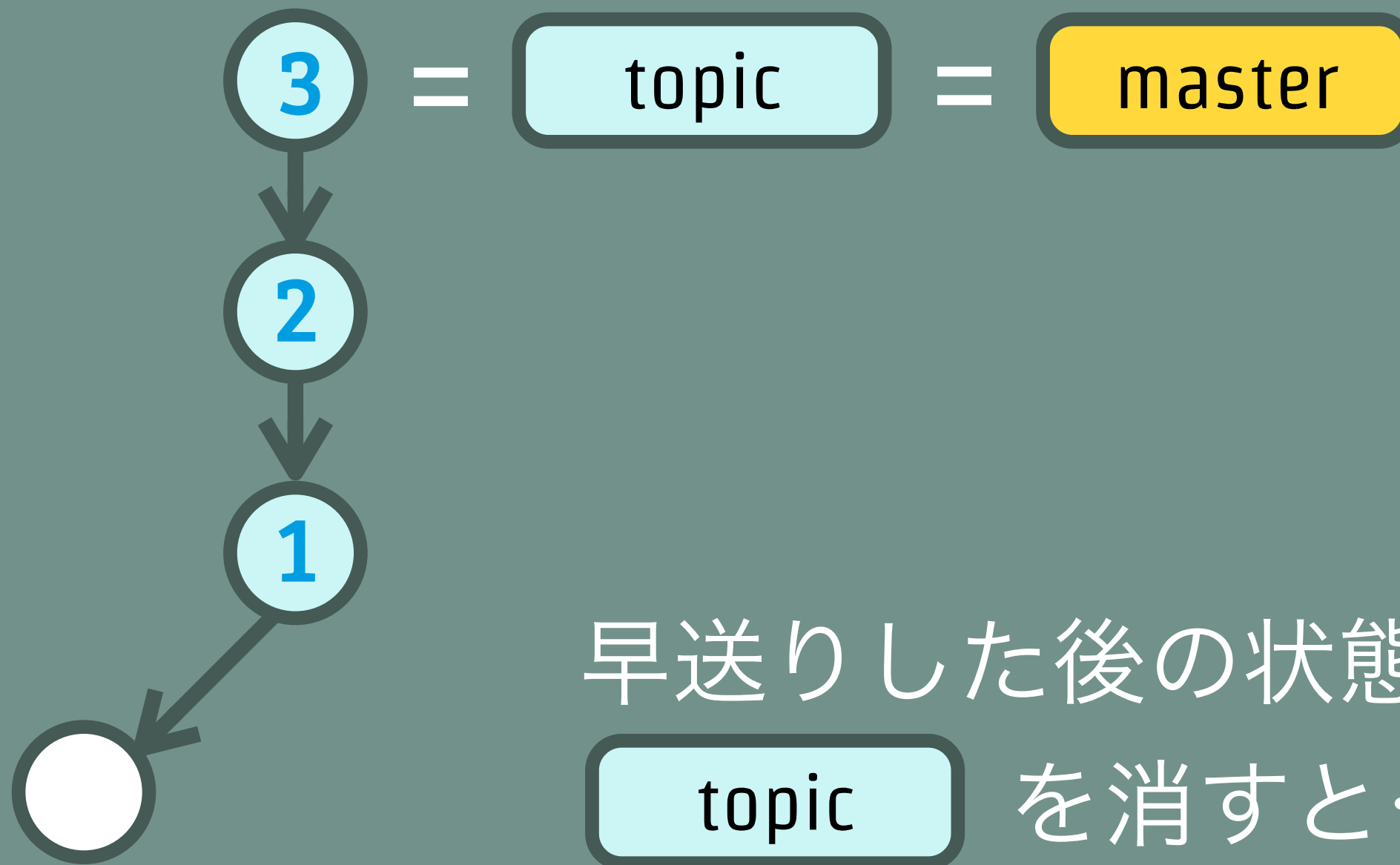
訳： 早送りマージすんな！

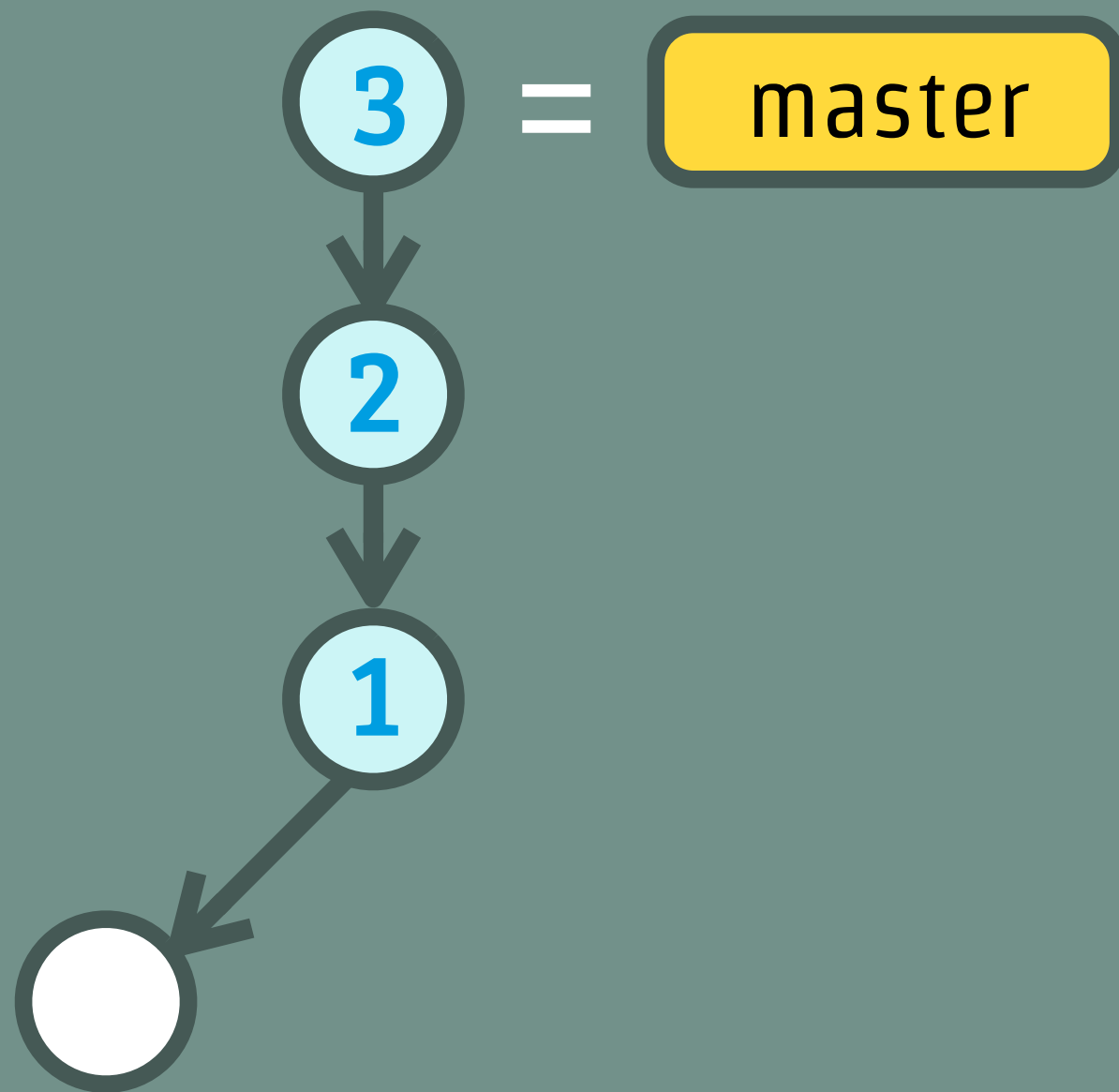
＼えっ／



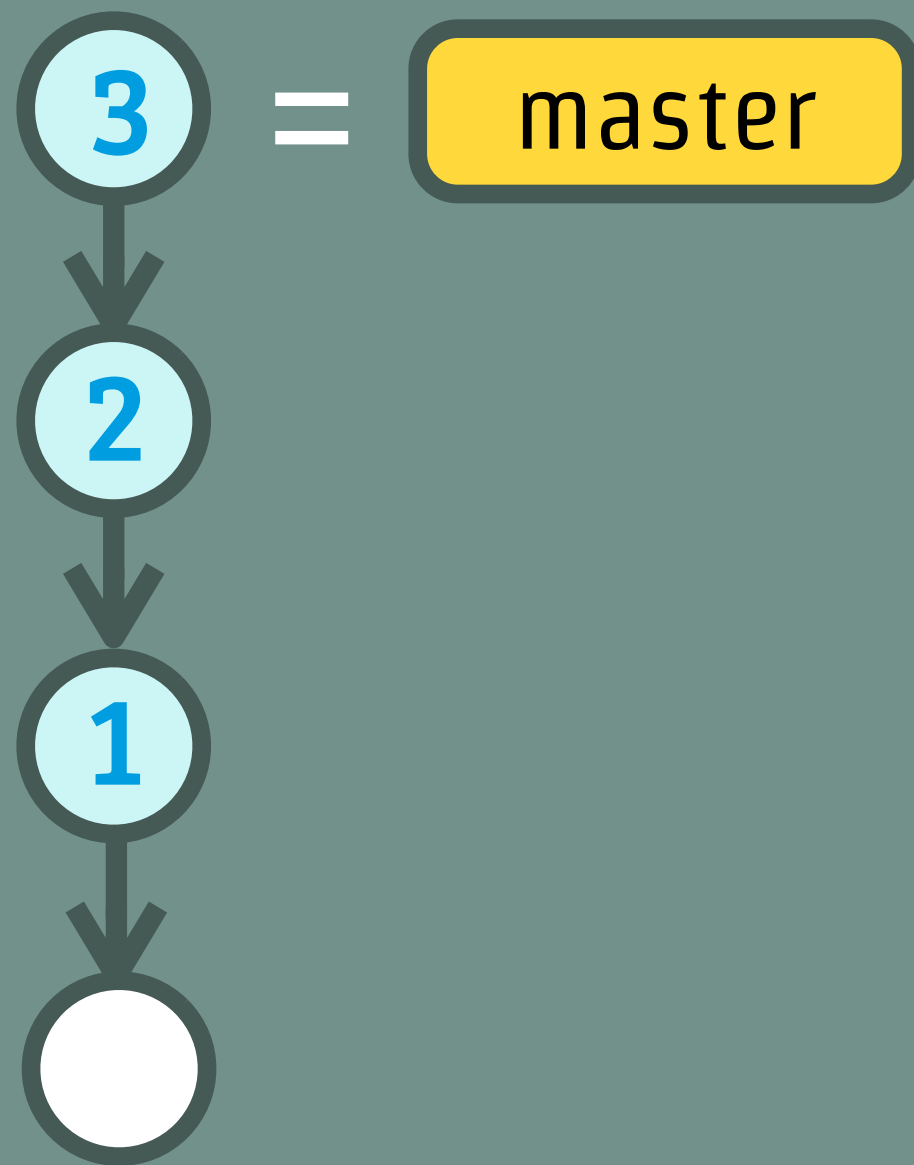
git

Why?

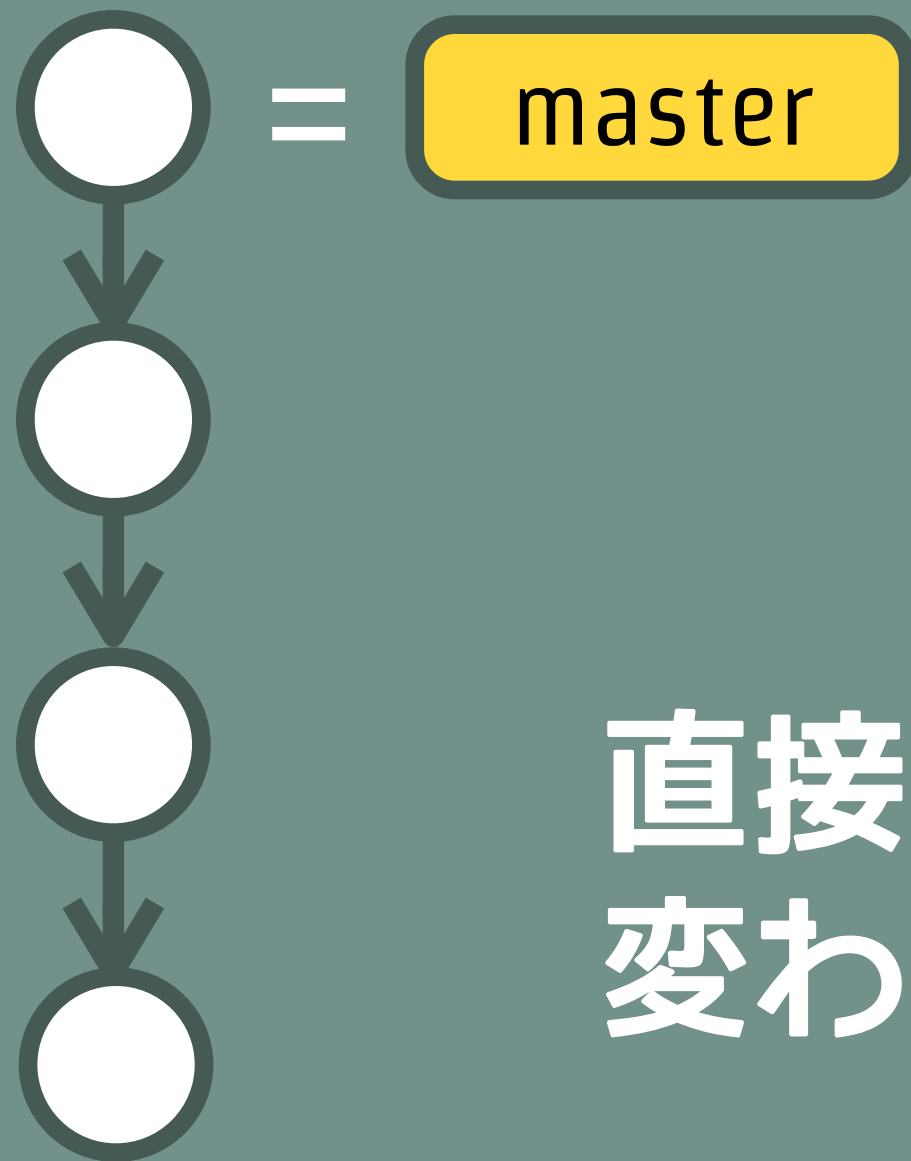




ん？

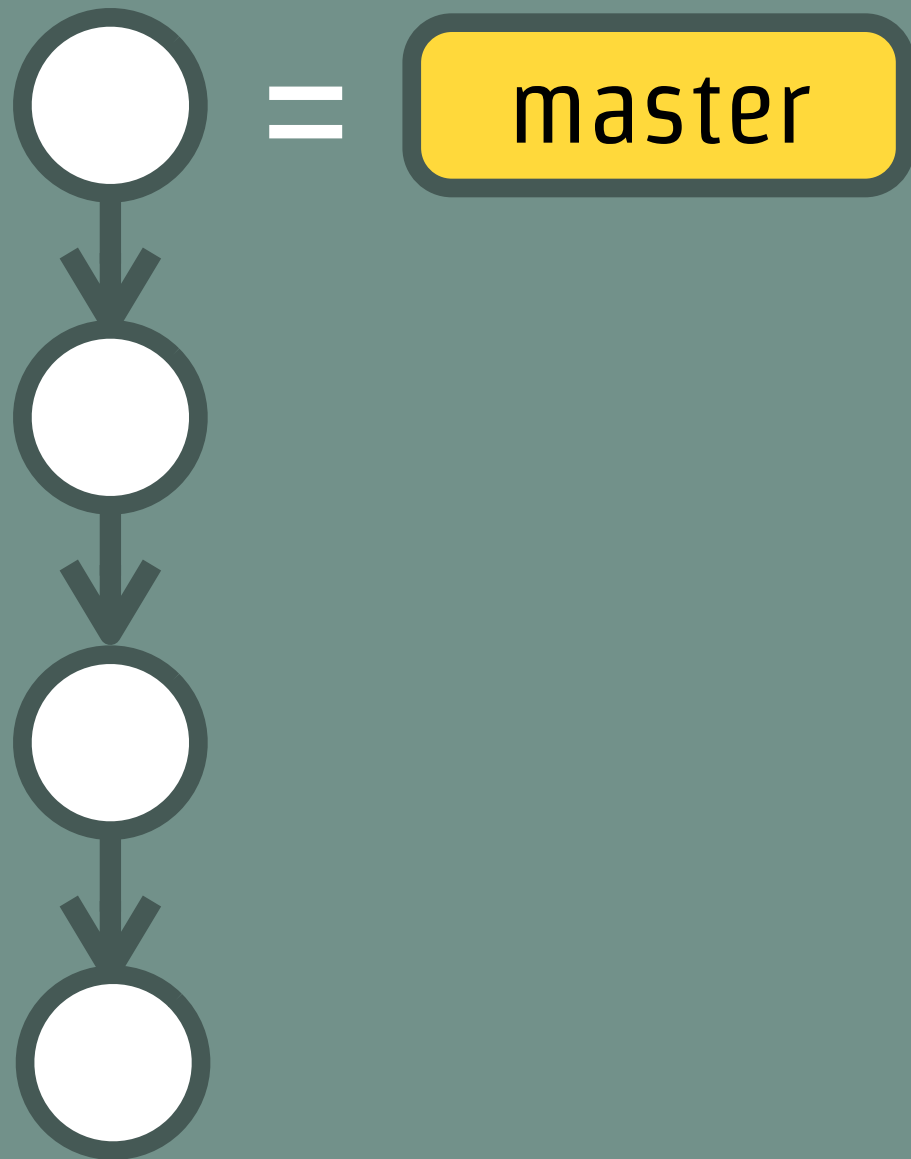


あれ？



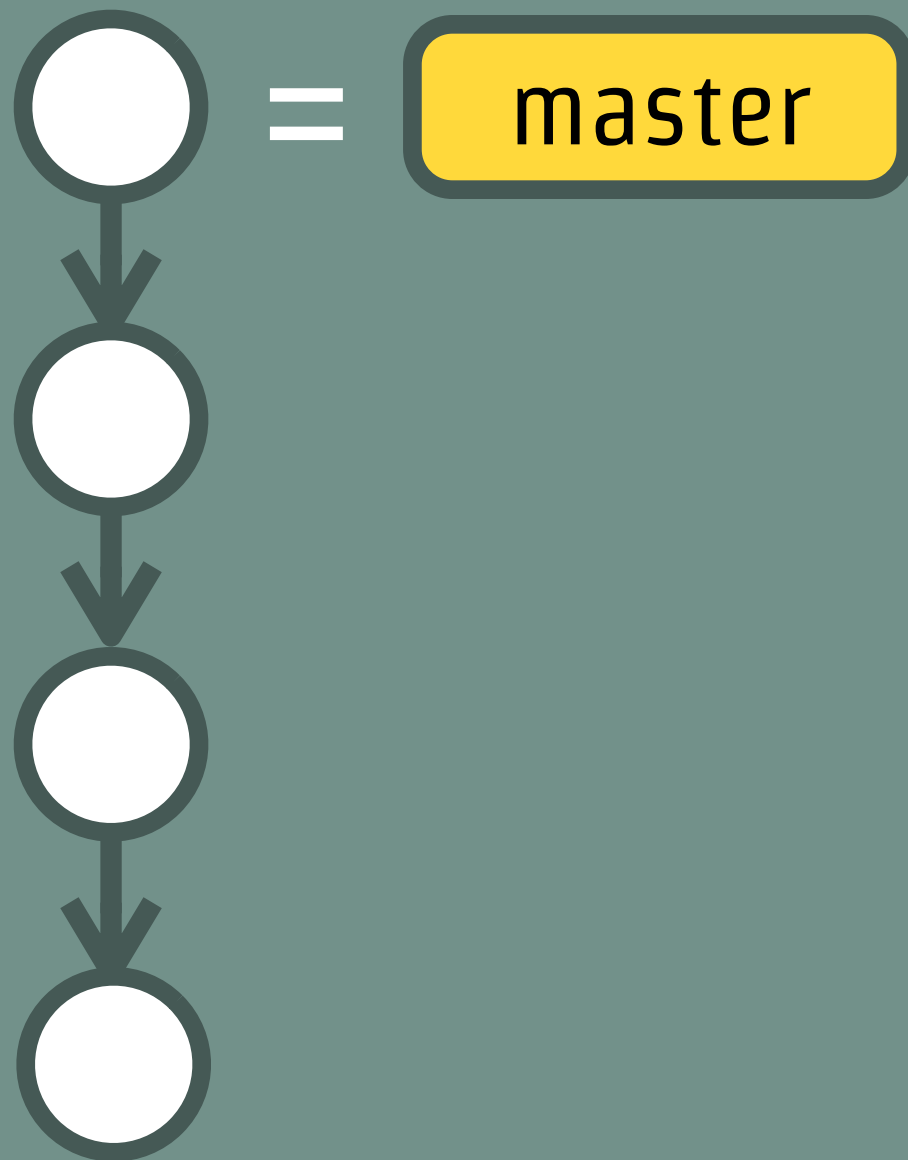
直接コミットしたのと
変わらない!!!

＼ **topic** の霊圧が消えた…!?! /

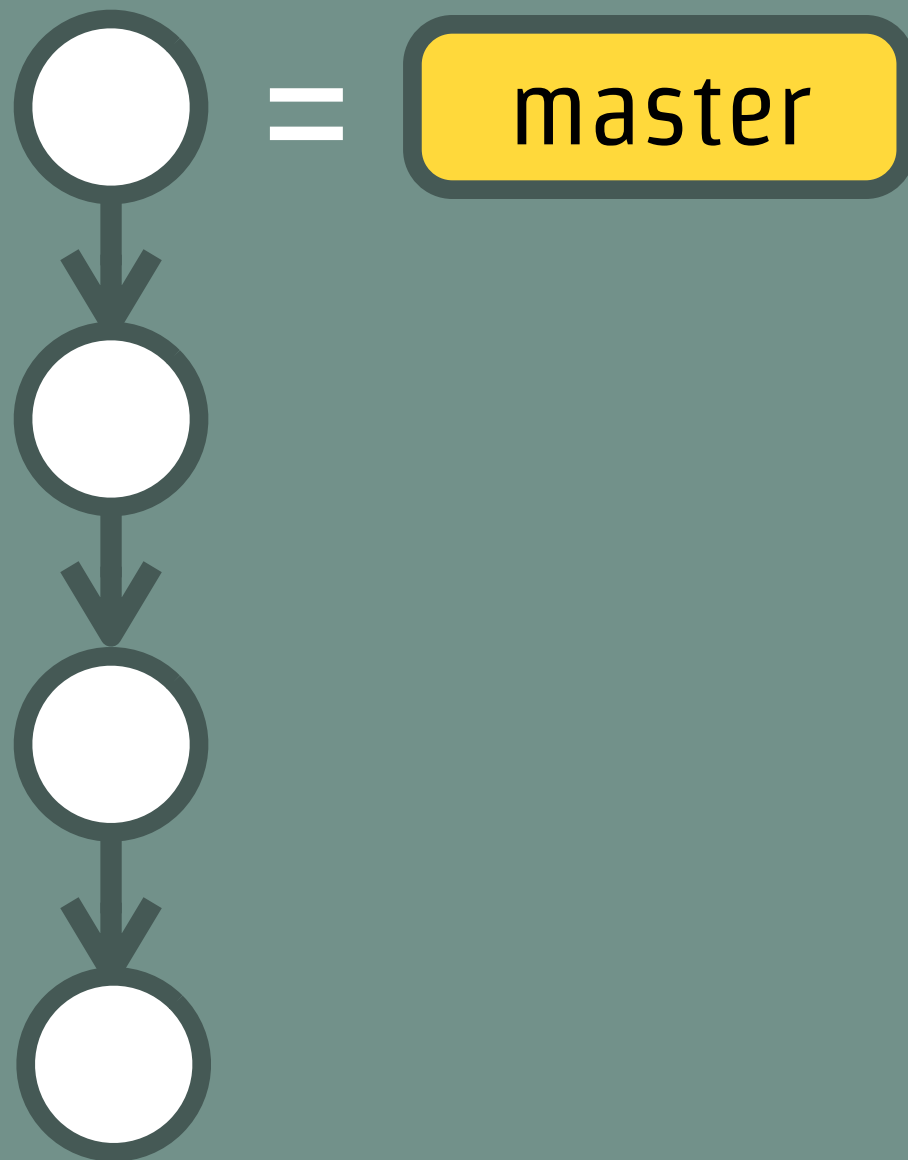


Fast-Forward の弊害①

「ブランチをマージした」という事実が
歴史（コミットグラフ）に残らない



おっと、間違っ
てマージしちゃった (>ω・)



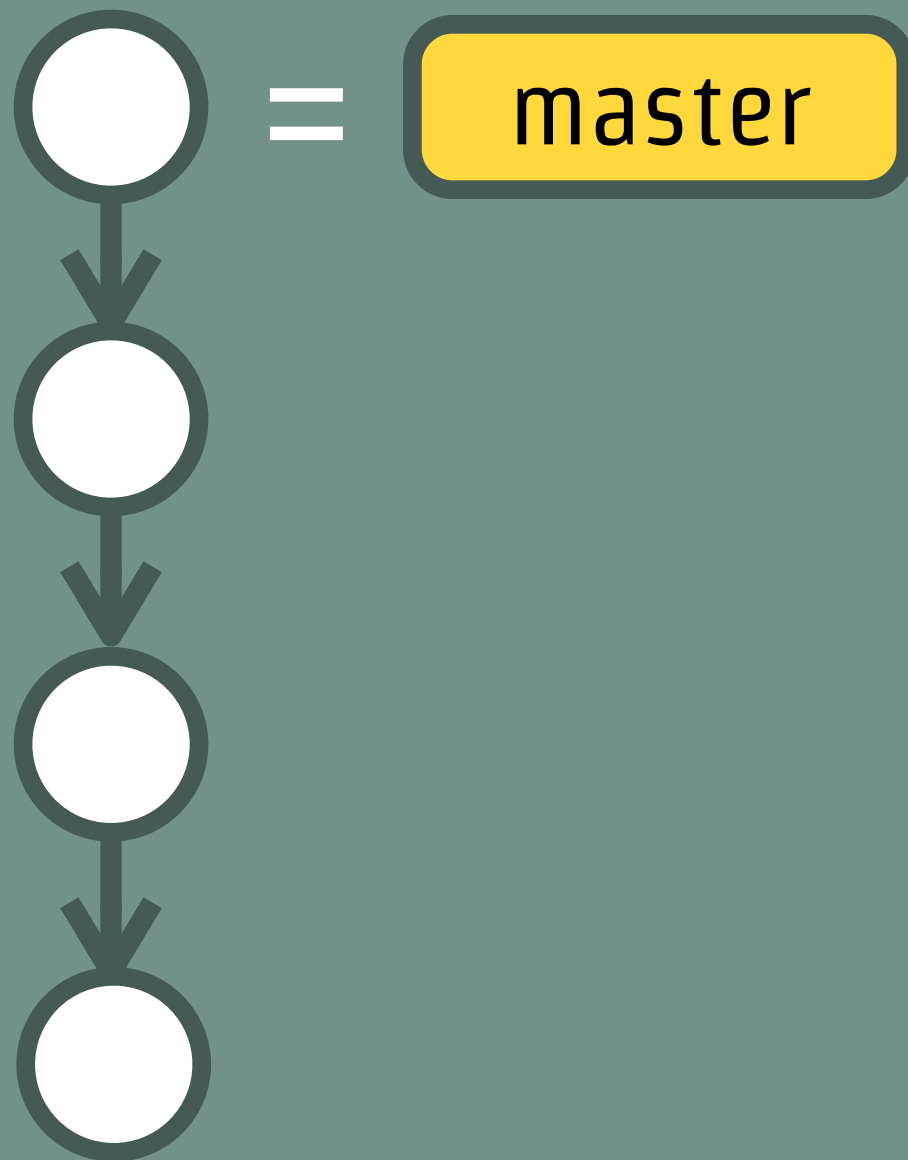
えーと、マージを
元に戻すにはっと...

git revert <commit>

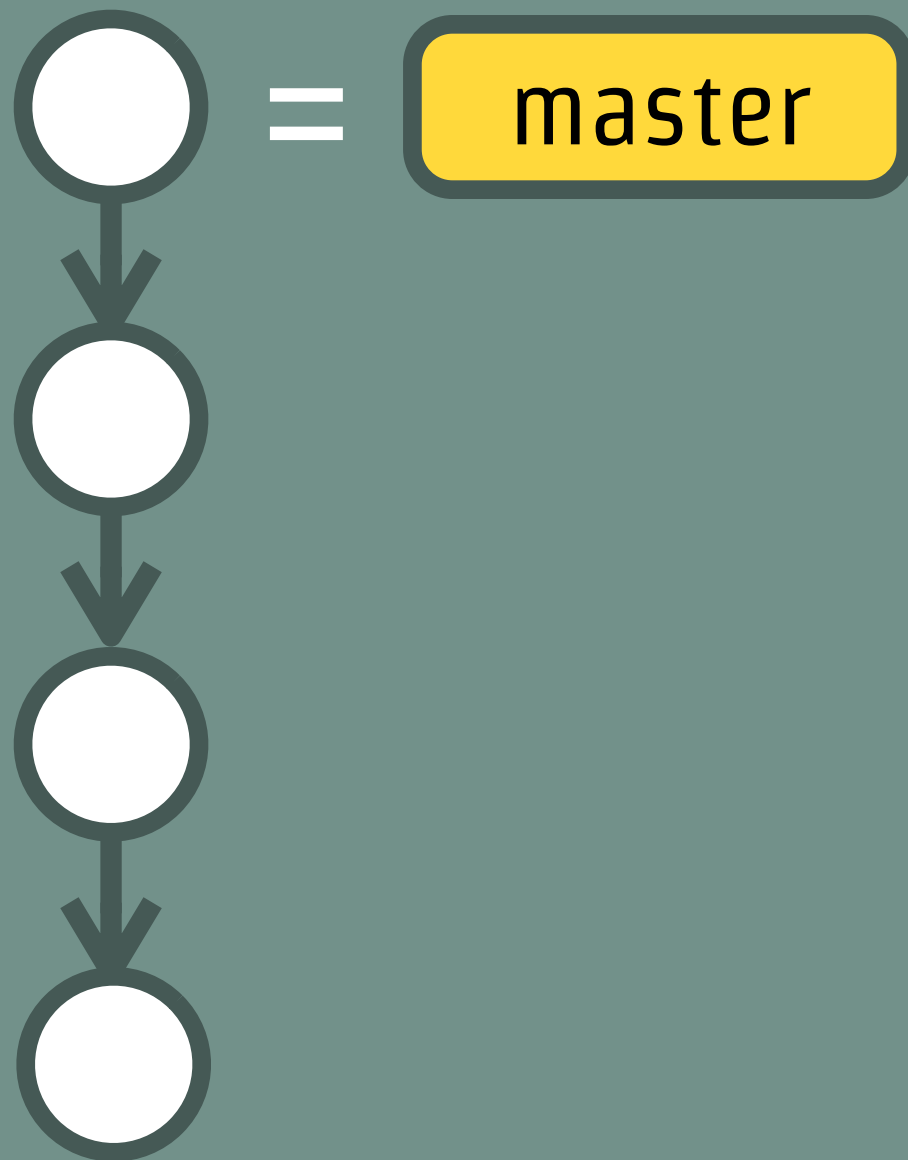
<commit> (リビジョン) のコミットを
取り消すためのコミットを作る

git reset --hard <commit>

<commit> (リビジョン) 時点の状態まで
完全に巻き戻す



おk おk
コミットを指定すればいいのか



で、どれが topic の
コミットだっけ……

Fast-Forward の弊害②

「ブランチのマージ」を
取り消しづらい

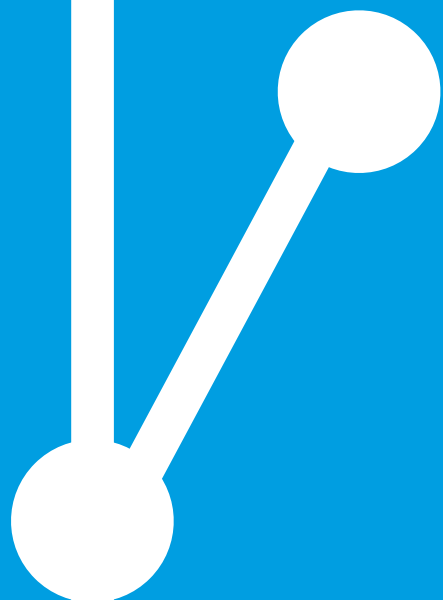
Point

マージの種類を意識して
マージしましょう！

二種類のマージ

- ・ **Fast-Forward** = 早送り
 - ・ コミットグラフ的に結果が同じなら同じコミットまで進めてしまうこと
 - ・ 「マージした」という記録が残らない
 - ・ マージを取り消しづらい
- ・ **Non Fast-Forward** = 普通のマージ
 - ・ Git ががんばって計算するマージ

リベースの功罪



#3

git 初心者が陥りがちな罠

第1位

(個人的に)

リベ————ス

git rebase

ブランチを master に追従するときに
使ってます!! マージはなんか怖いし...

rebase すると、コミットログが
キレイになって見やすいよね!!

merge よりも rebase の方が
マージコミットもなくて楽でしょ?

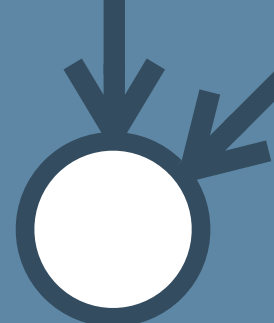
Question

そもそも rebase が
なにをするのか
理解していますか？

2 = master

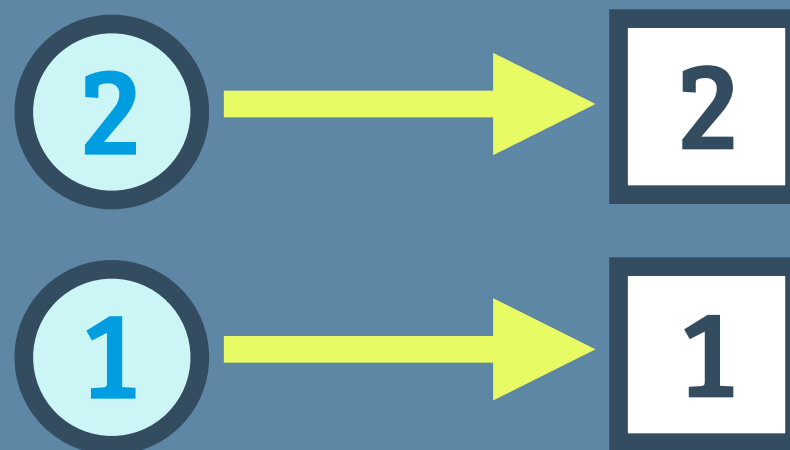


2 = topic

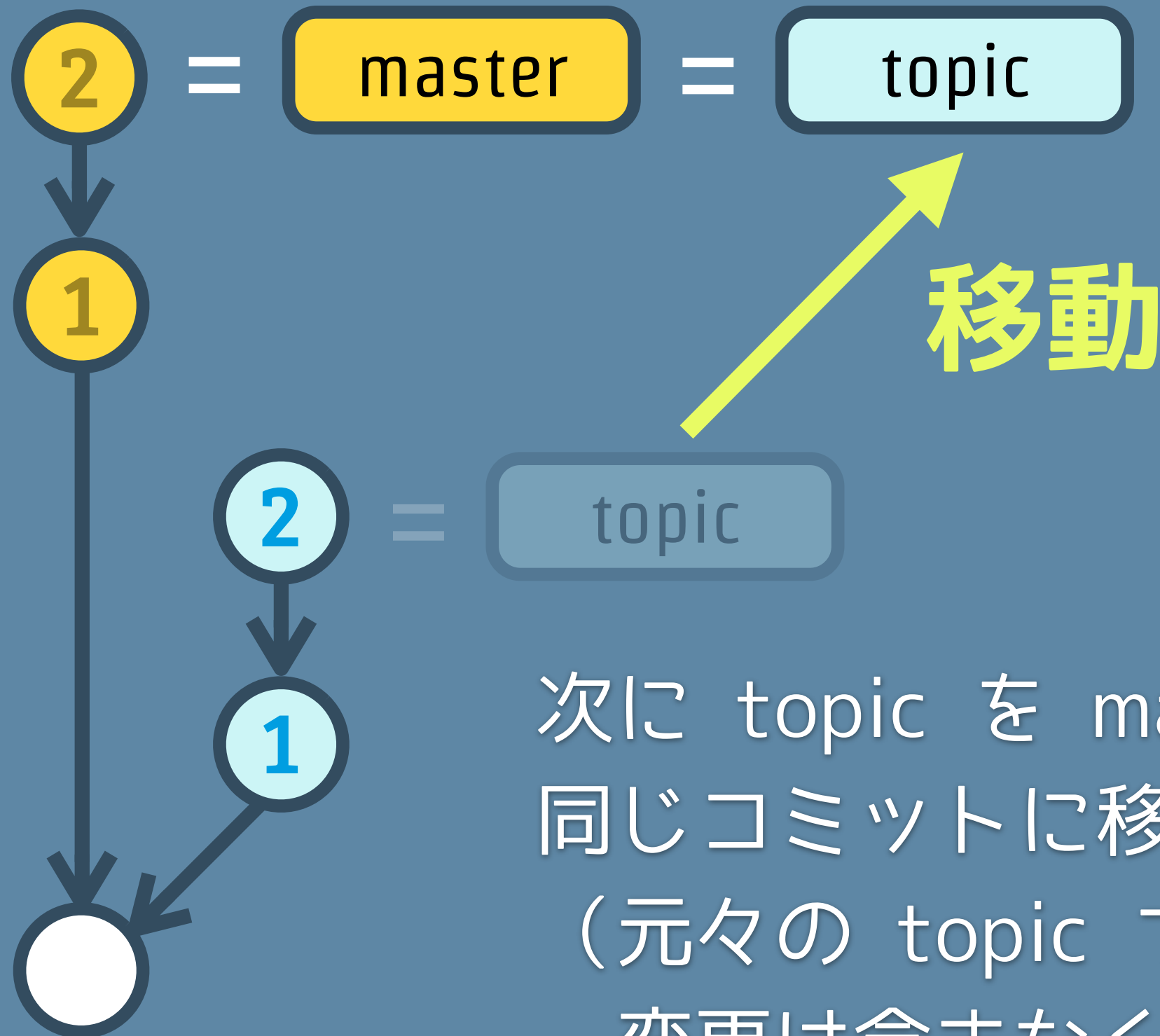


この状態から **topic** で
`git rebase master` すると...

topic

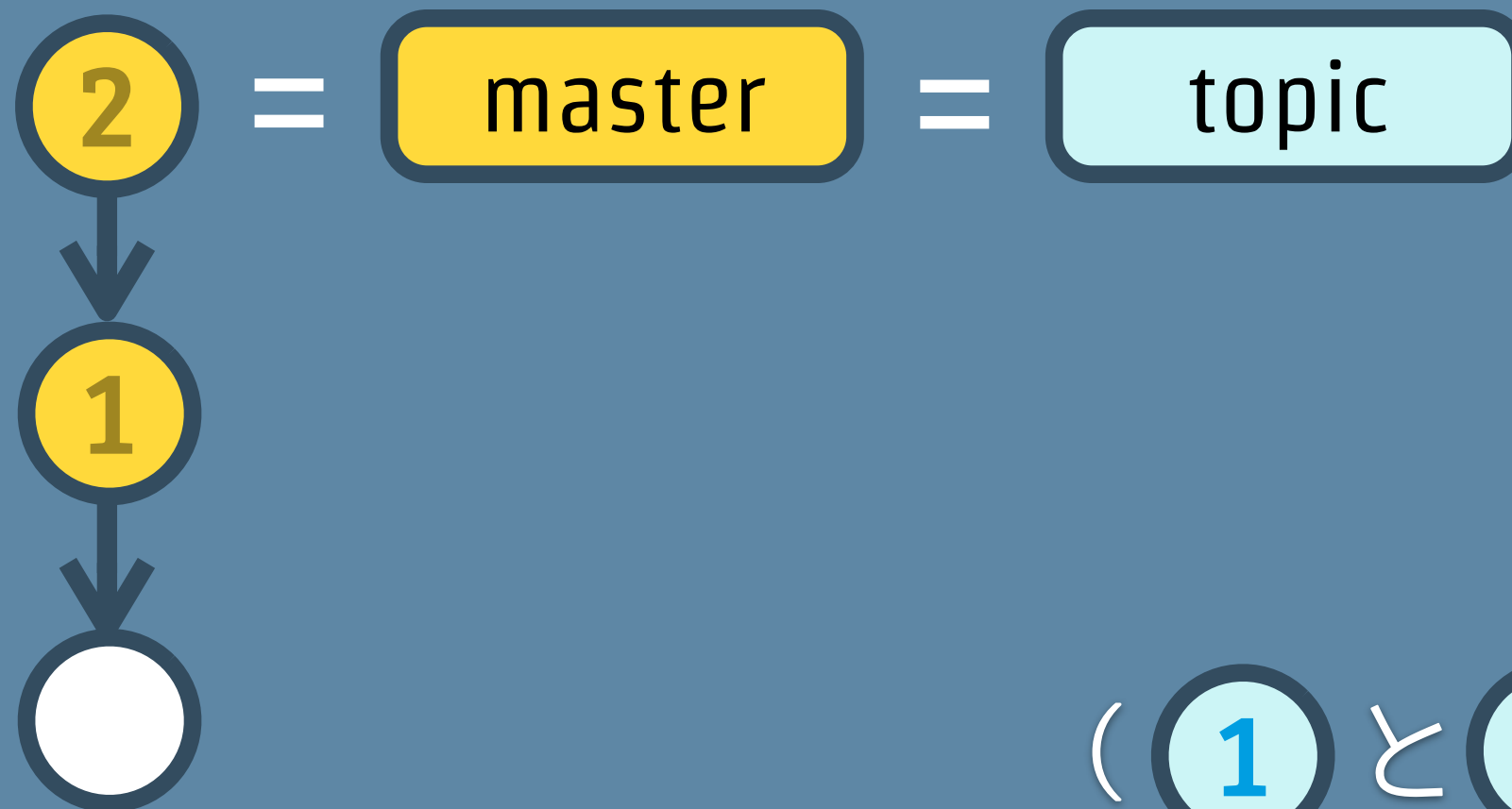


まず、コミットの変更内容を
それぞれパッチにする



次に topic を master と
同じコミットに移動させる
(元々の topic ブランチでの
変更は含まなくなる)

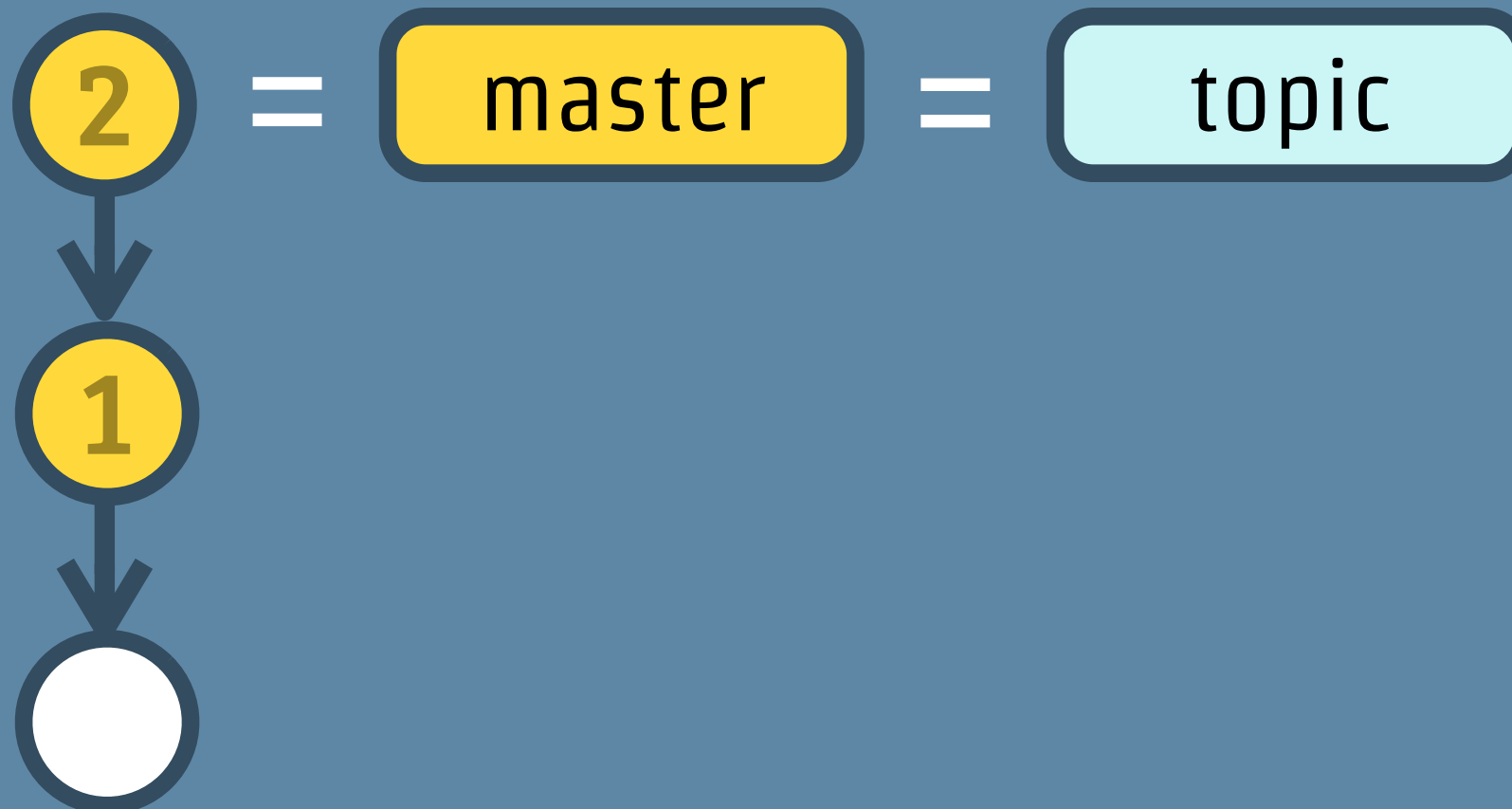
ここまで来たら、作っておいた
パッチを1つずつ順に適用して
コミットしていく

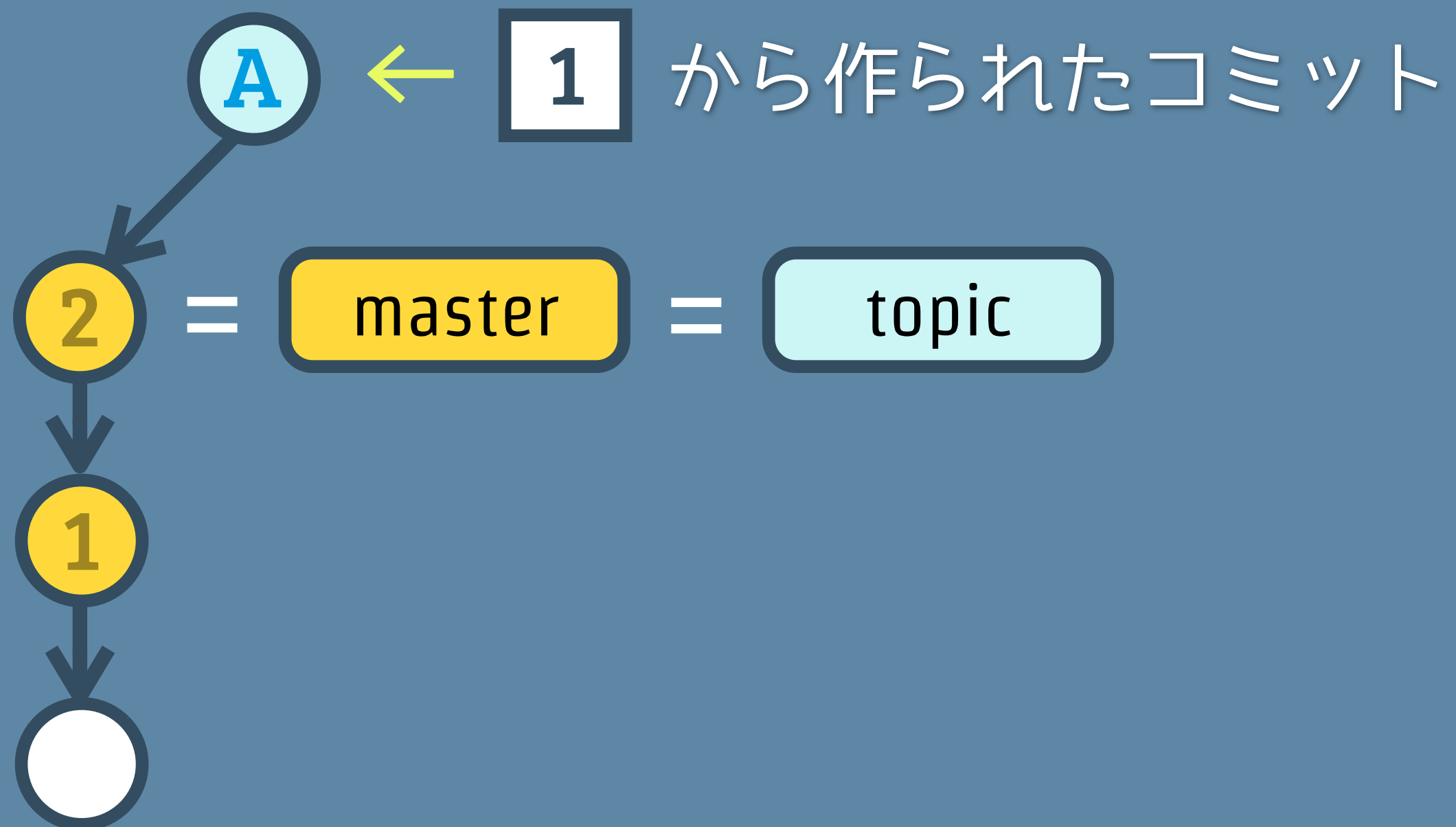


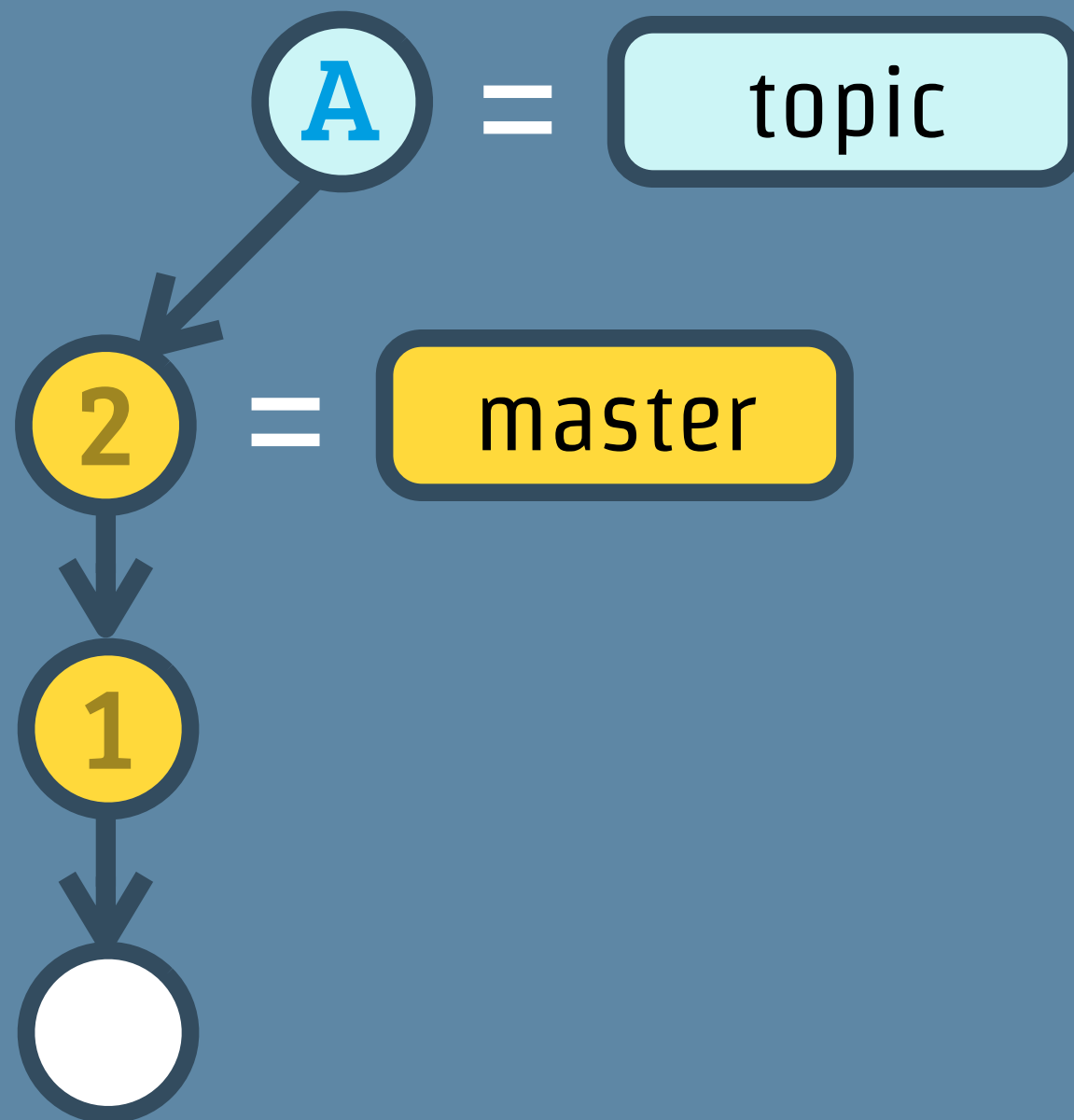
(**1** と **2** は省略)

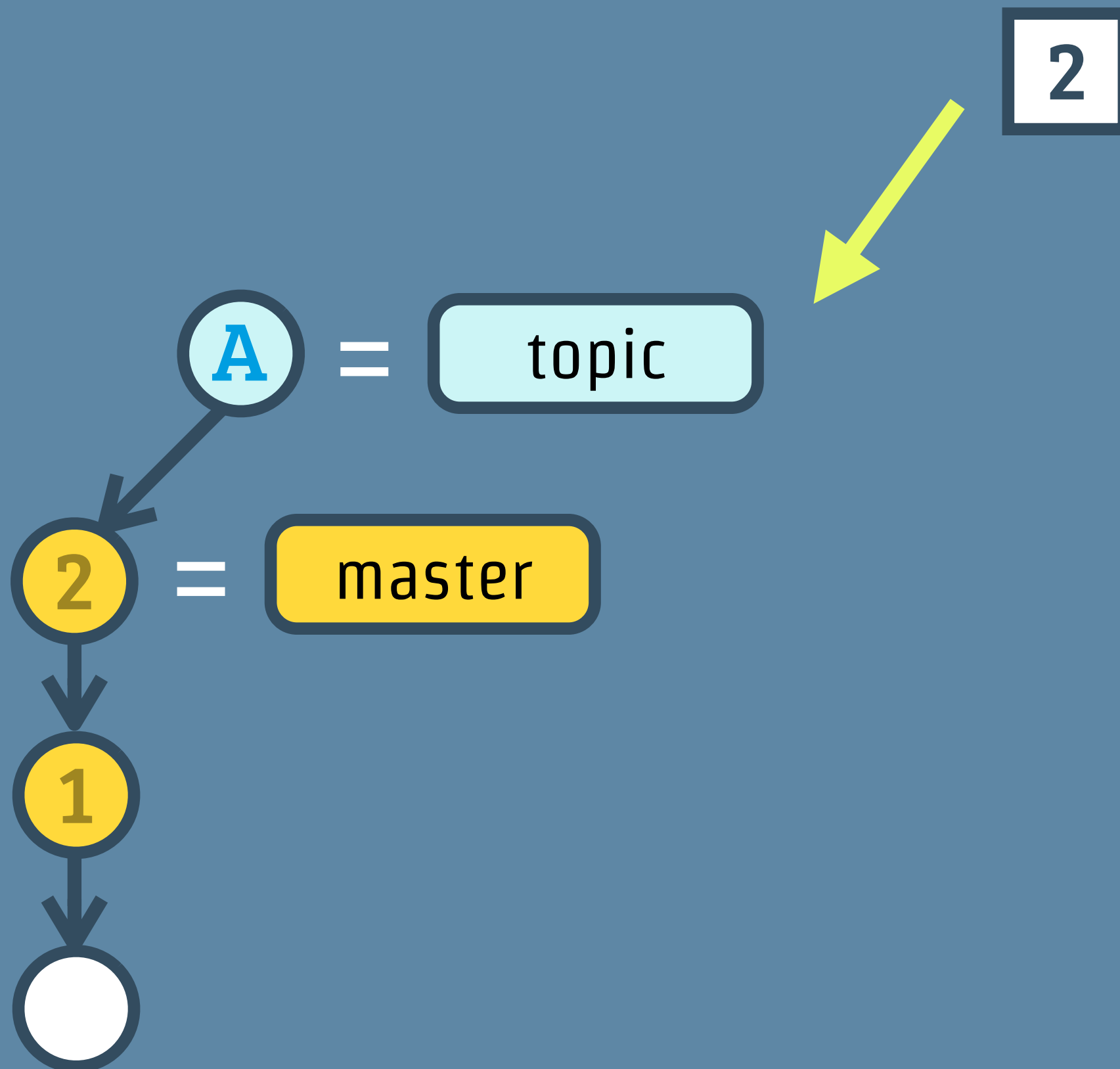
差分を適用してコミット

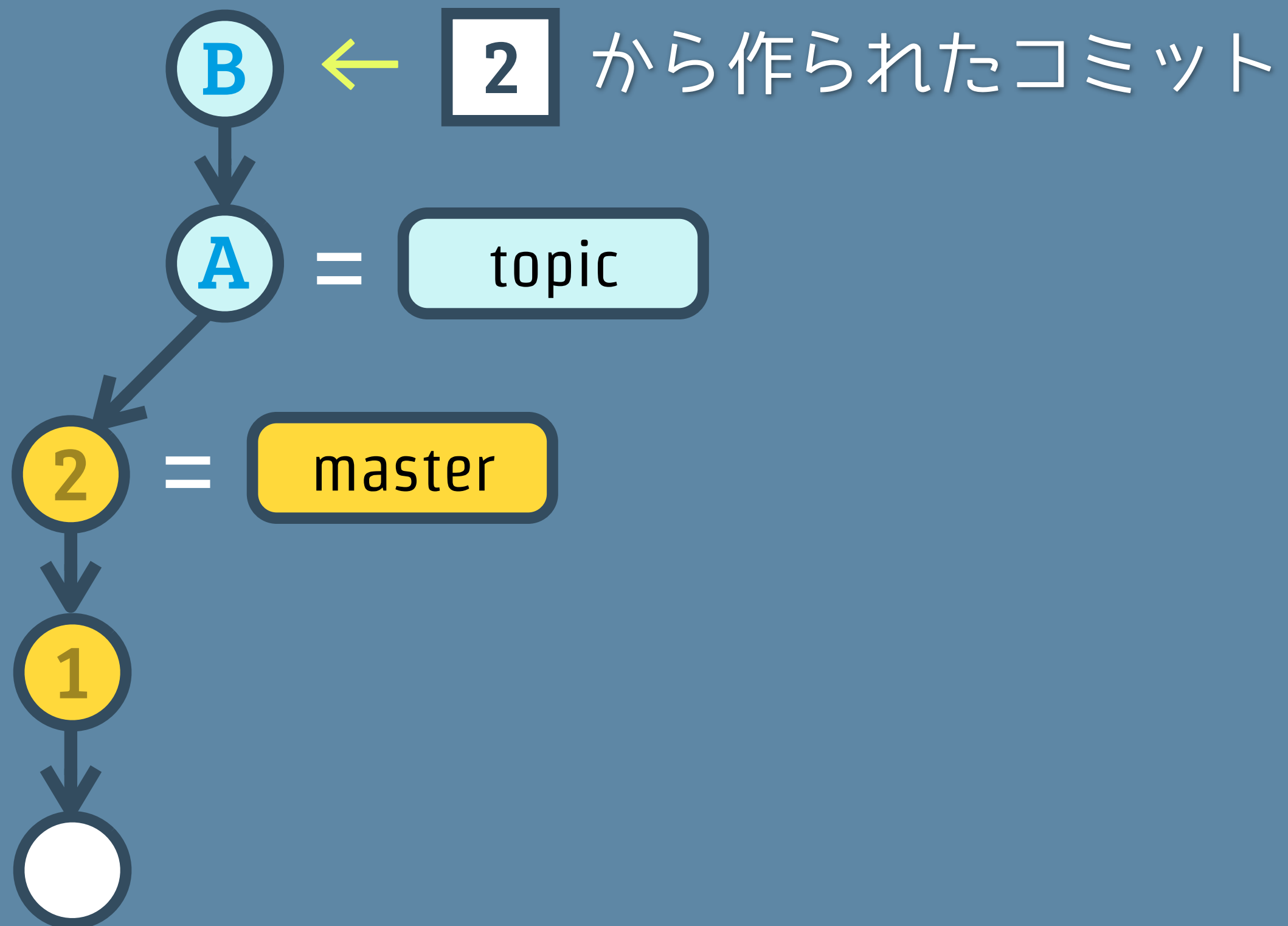
1

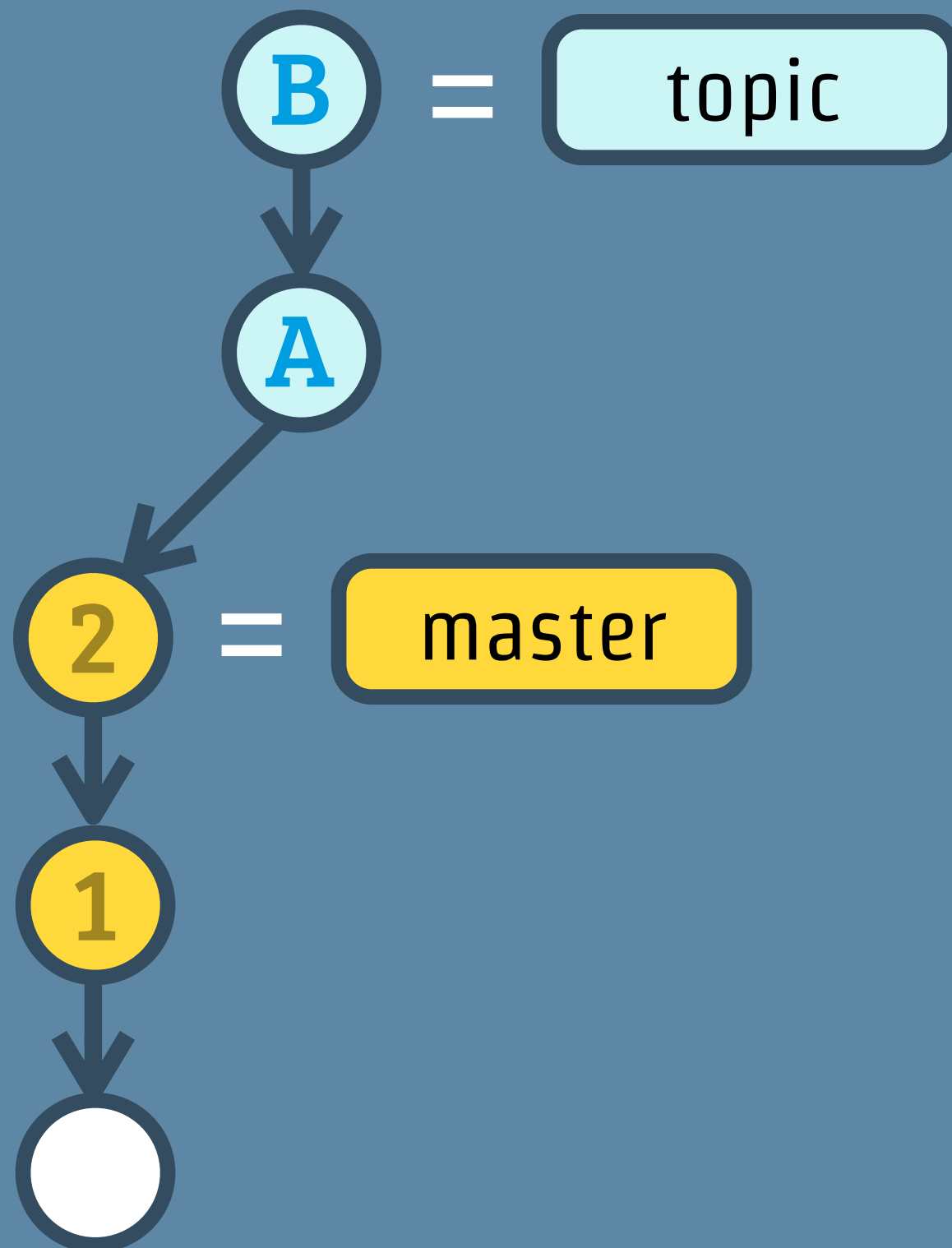




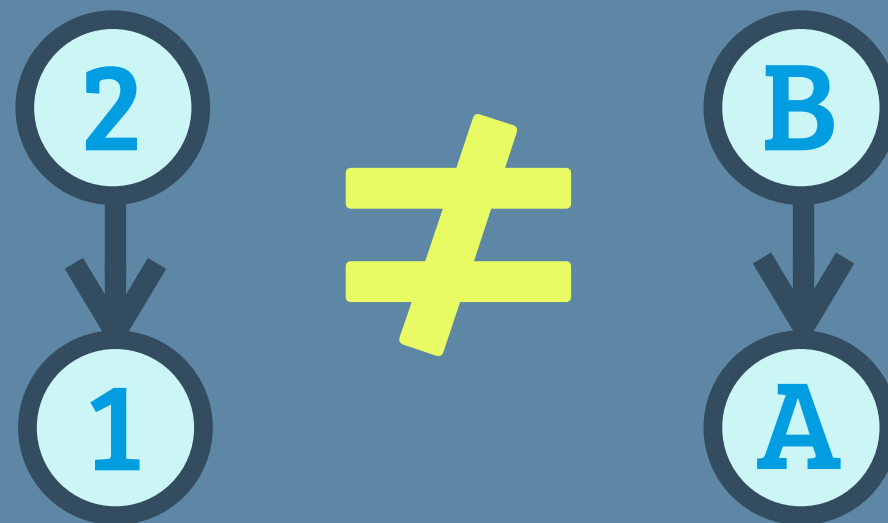






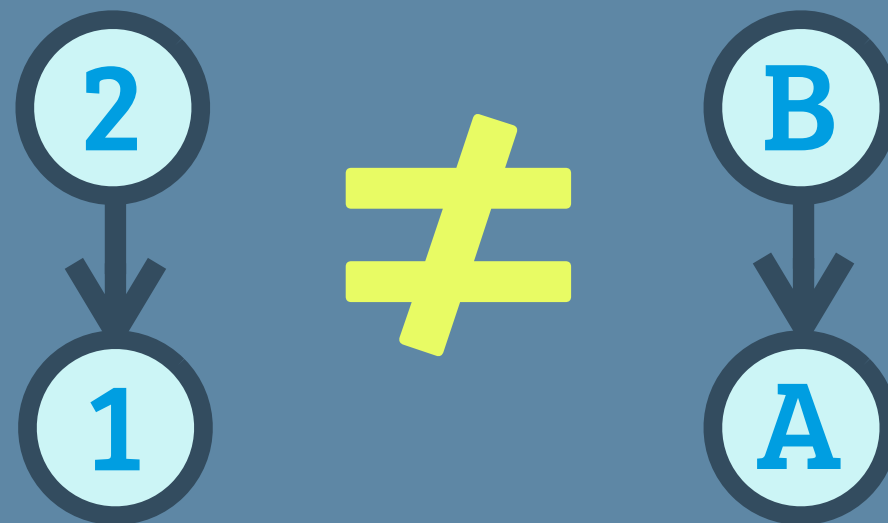


重要



rebase で作られたコミットは
元のコミットと同じ内容だけど
別のコミットになります!!

重要



本当？

rebase で作られたコミットは
元のコミットと同じ内容だけど
別のコミットになります!!

コミットに入ってる情報

リビジョン (SHA-1 ハッシュ)

例: 23cdd334e6e251336ca7dd34e0f6e3ea08b5d0db

Author (コミットを作成した人)

例: オープンソースプロジェクトにパッチを送った人

Committer (コミットを適用した人)

例: 受け取ったパッチを取り込んだ人

ファイルのスナップショット (tree)

コミットで変更されたファイルを含むツリー (説明は省略)

1つ前のコミットのリビジョン

例: 4717e3cf182610e9e82940ac45abb0d422a76d77

コミットに入ってる情報

リビジョン (SHA-1 ハッシュ)

例: 23cdd334e6e251336ca7dd34e0f6e3ea08b5d0db

Author (コミットを作成した人)

例: オープンソースプロジェクトにパッチを送った人

Committer (コミットを適用した人)

例: 受け取ったパッチを取り込んだ人

ファイルのスナップショット (tree)

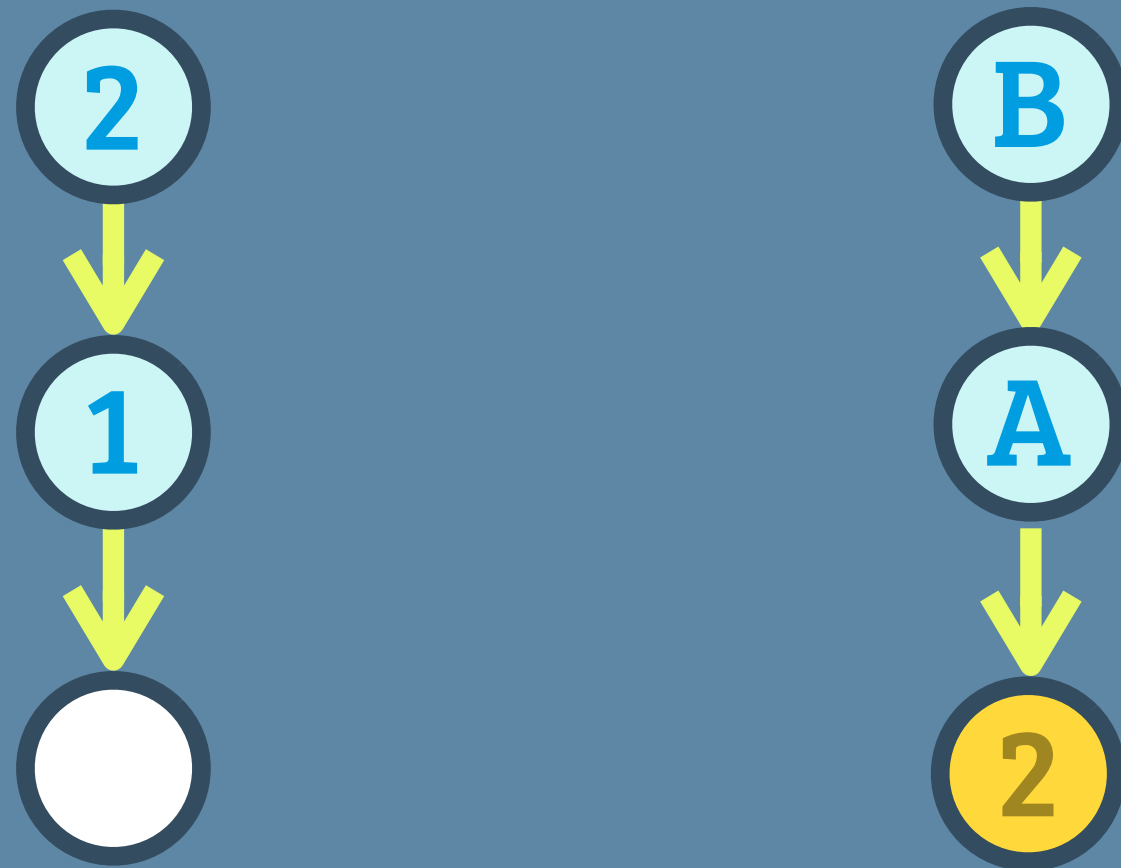
コミットで変更されたファイルを含むツリー (説明は省略)

1つ前のコミットのリビジョン

例: 4717e3cf182610e9e82940ac45abb0d422a76d77



1つ前のコミットが違う!!



1つ前のコミットが変わると
リビジョンも変わります
(だから別物になる)

Point

rebaseで作られるコミットは
元のコミットとは別物



別物でも、同じ変更が
されるから問題ない気が

git push origin topic

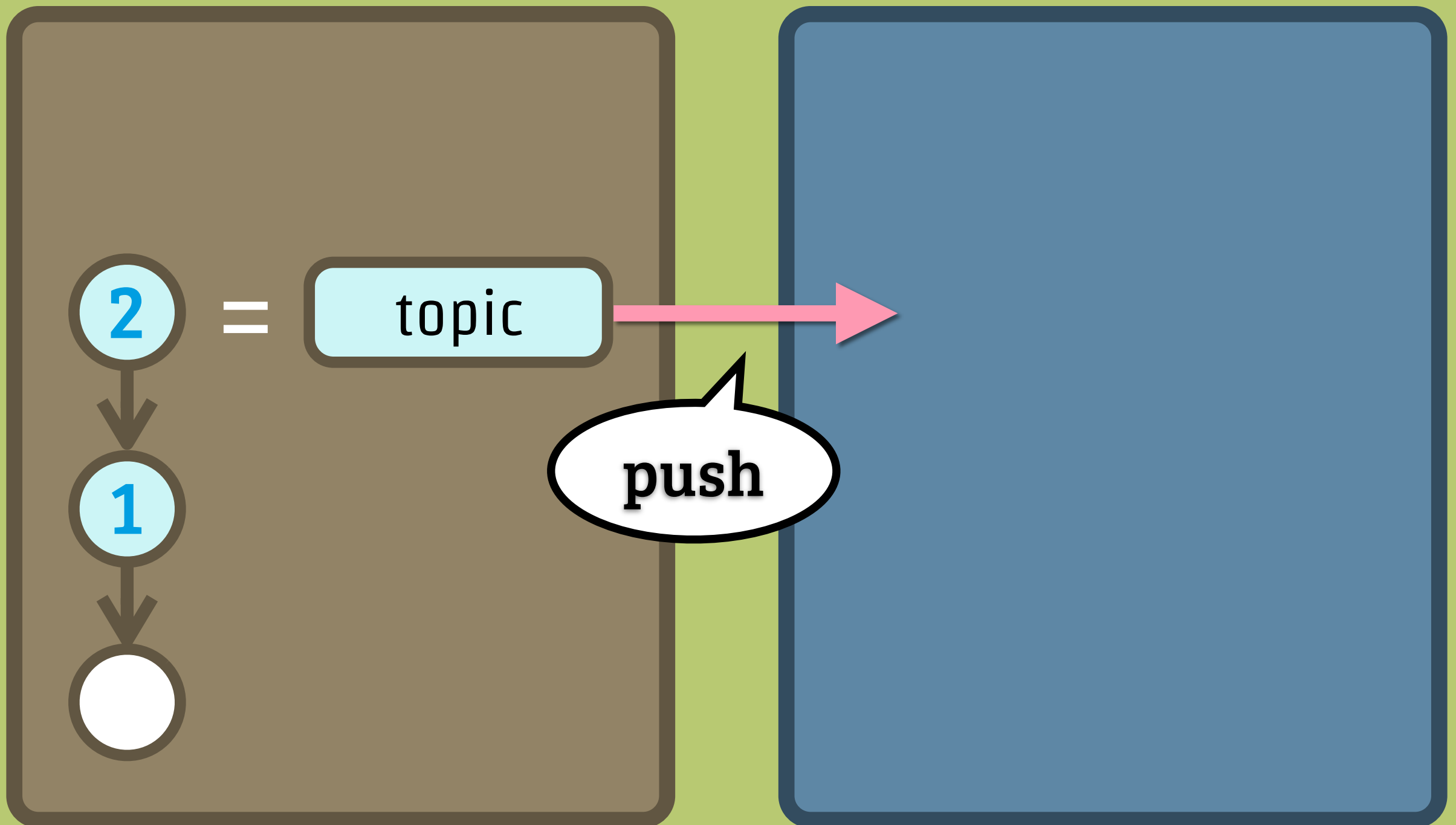
git push origin `topic`

`topic`

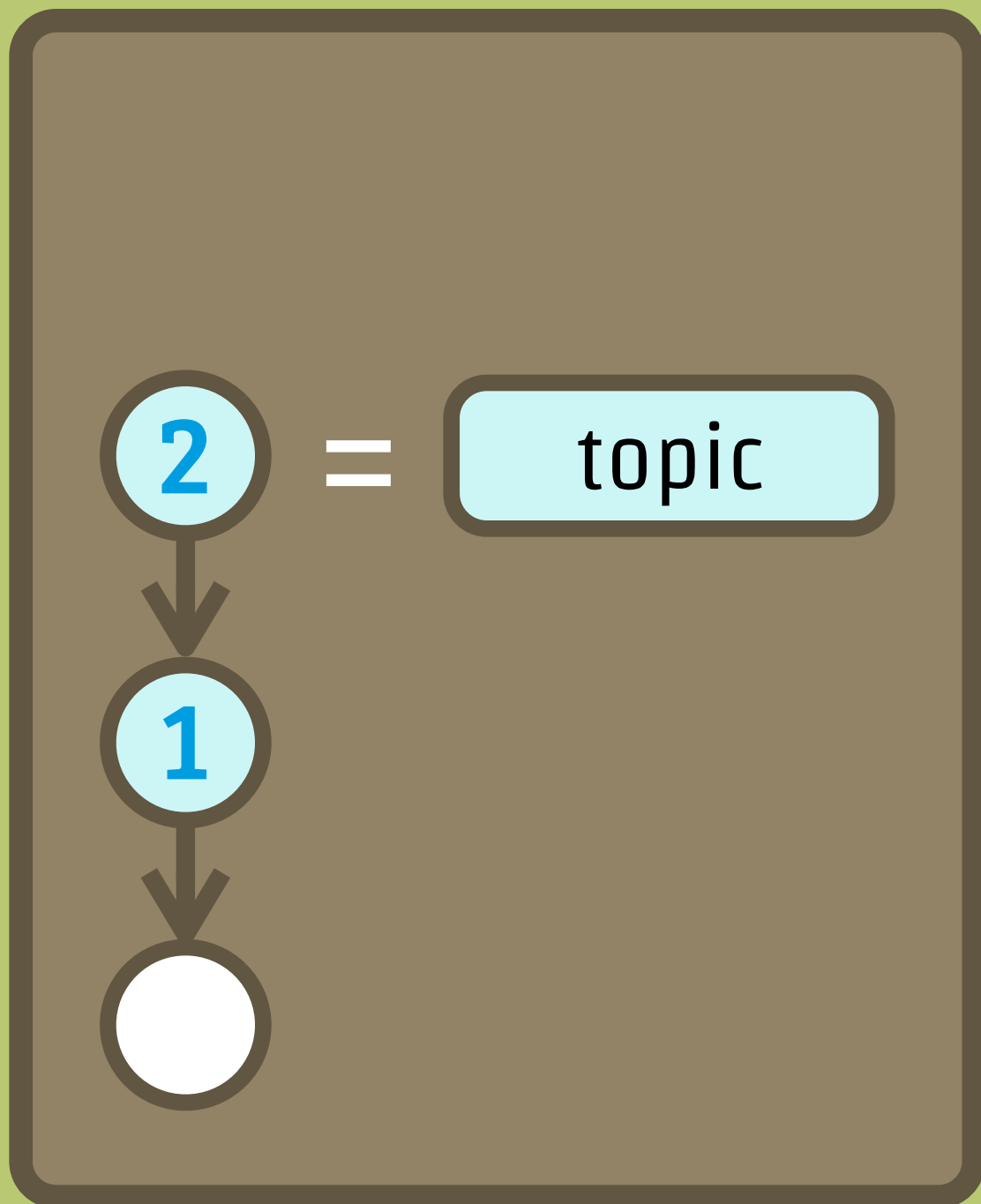
を origin (サーバー) に同期

local

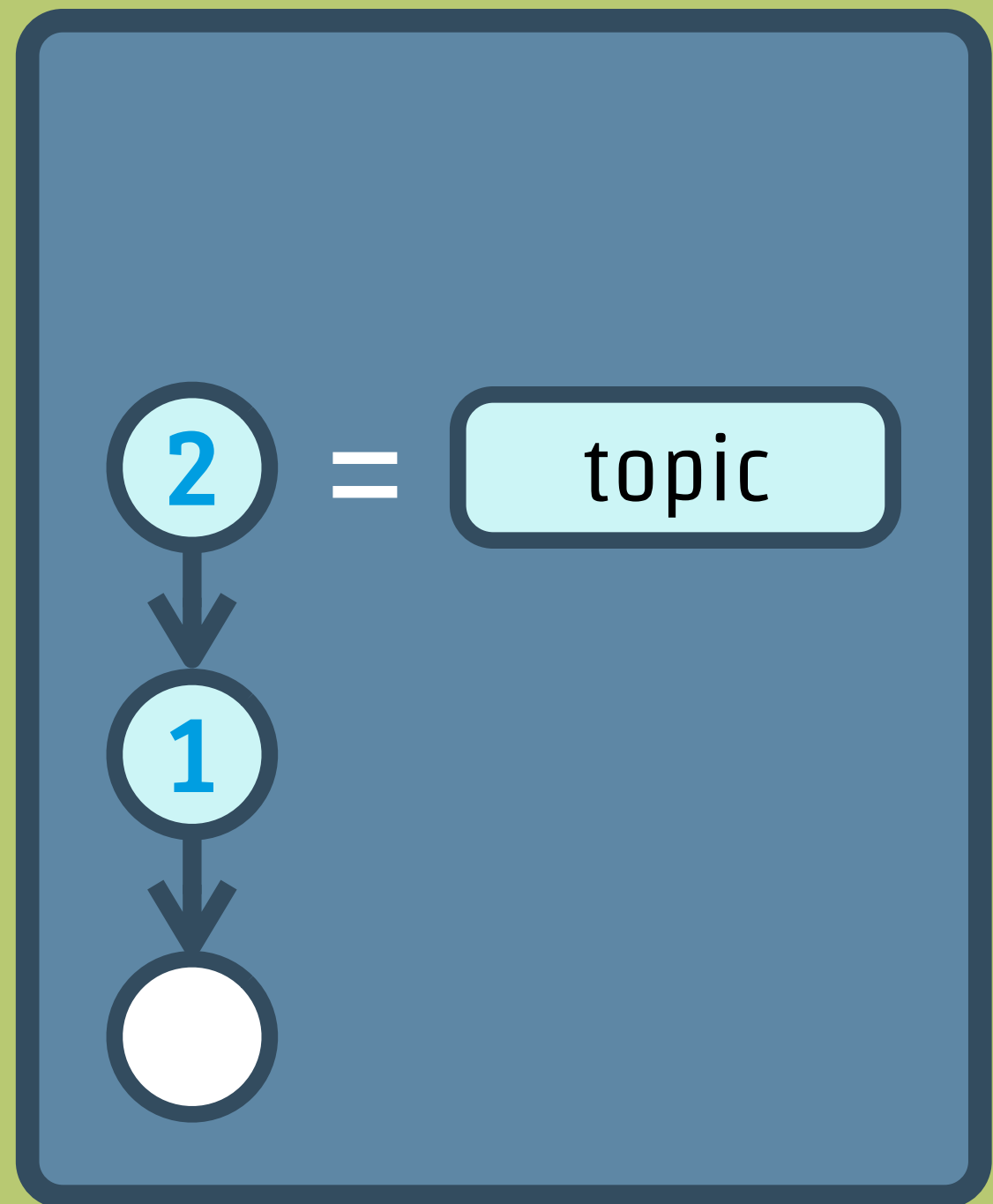
origin



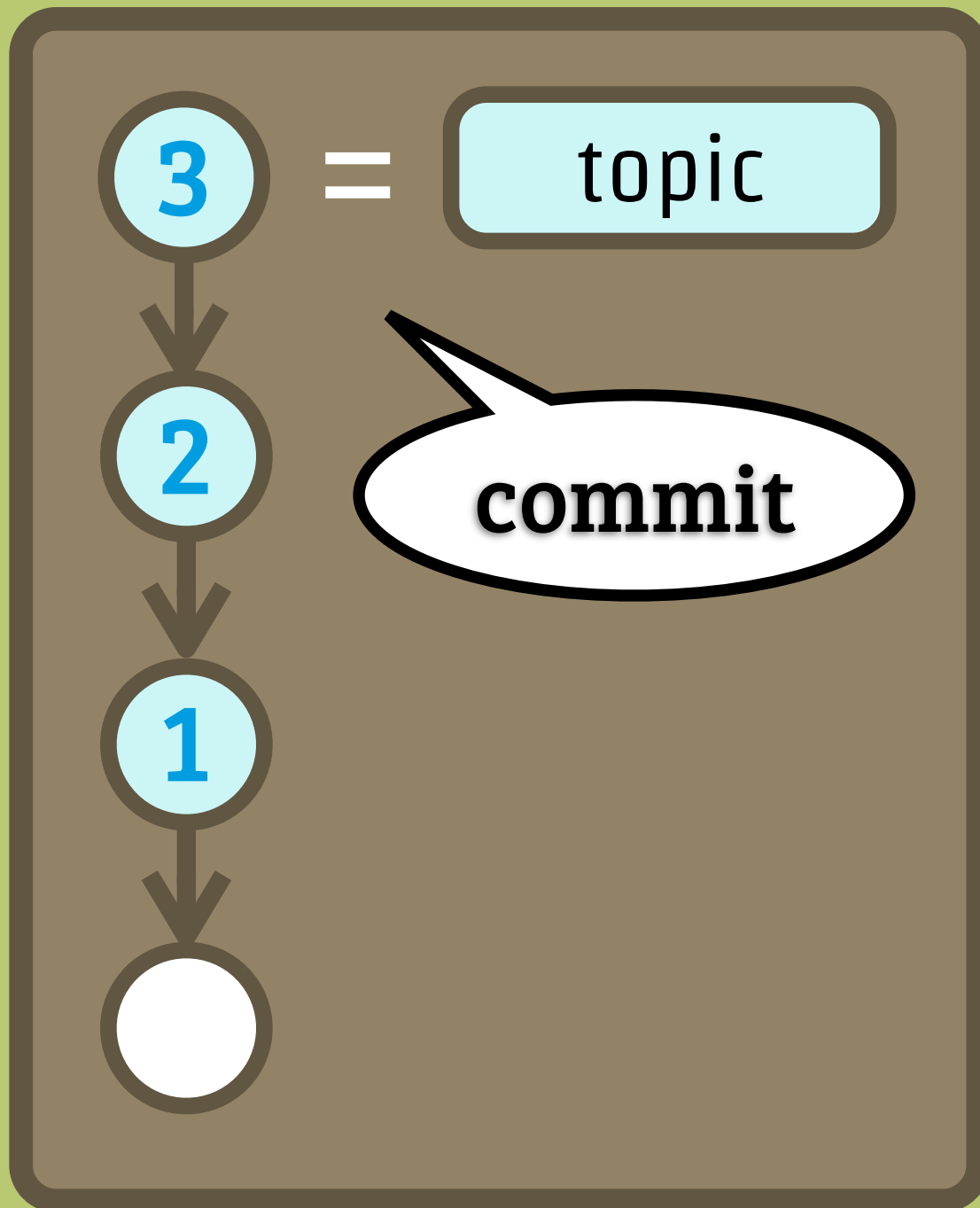
local



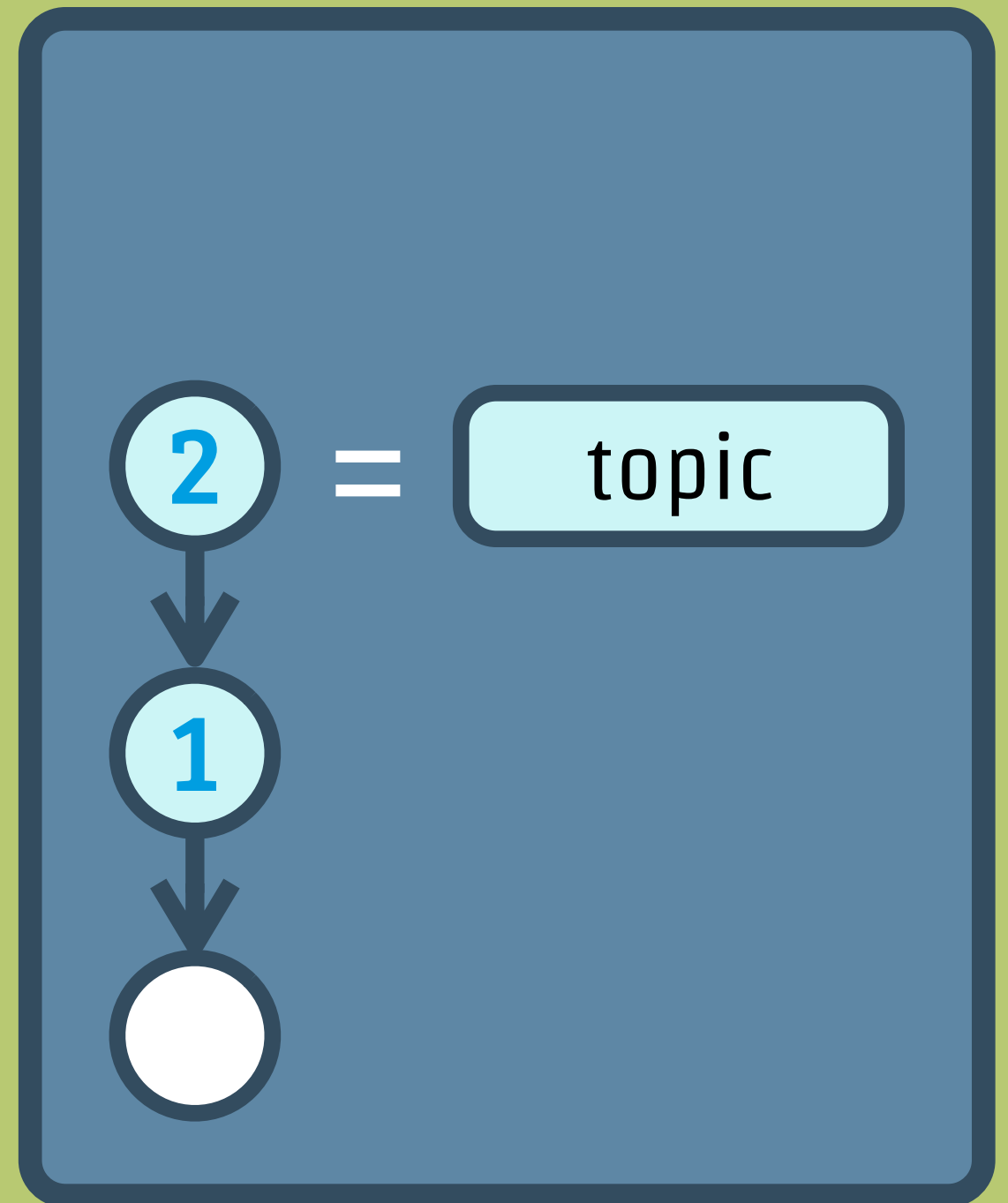
origin



local

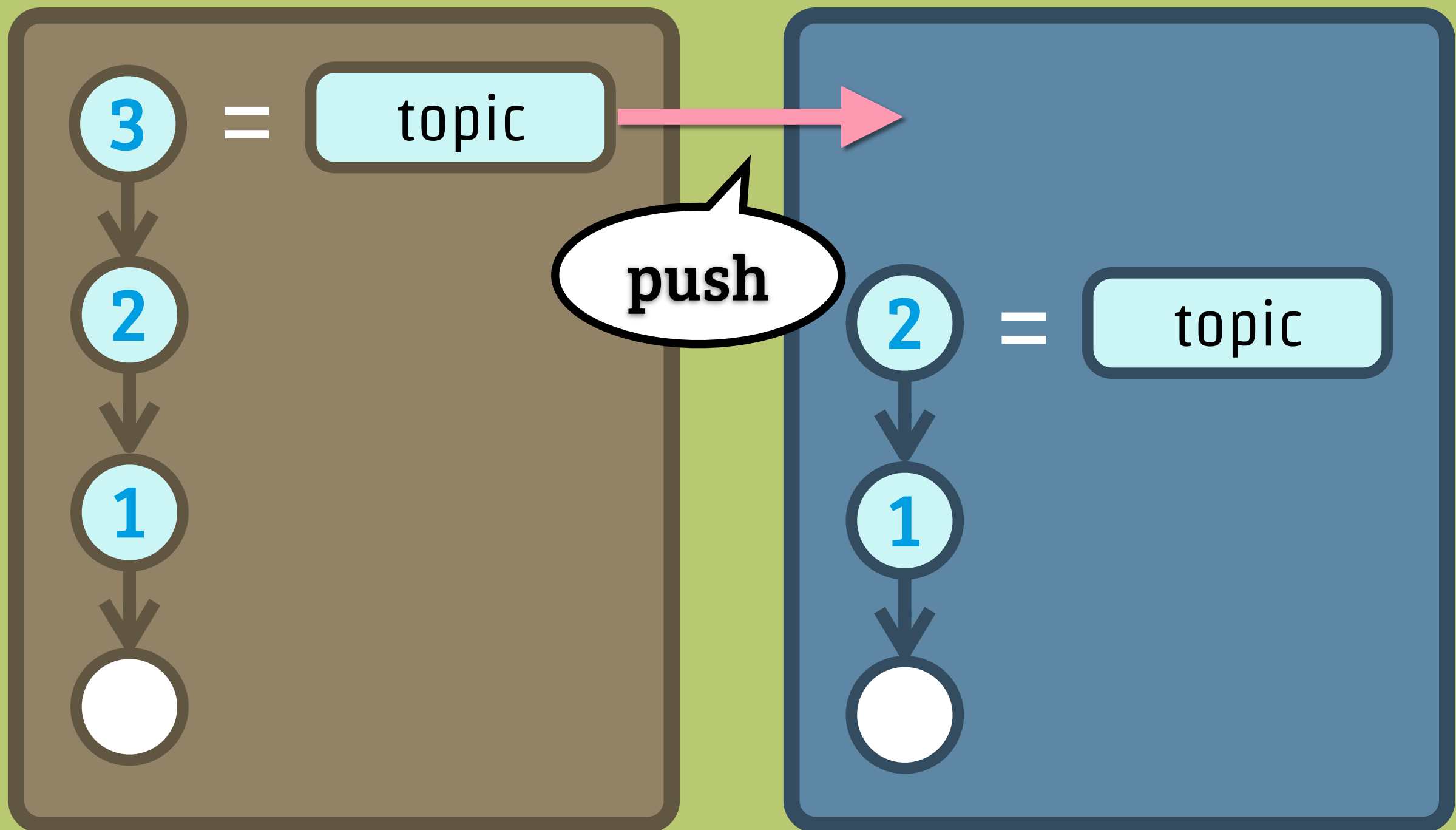


origin



local

origin

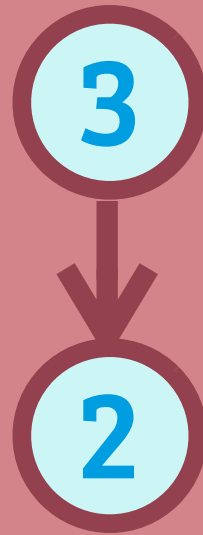


origin 「お、pushがきたぞ？
どれどれ、内容を見てみよう」

origin 「新しいコミットは1つだけか」

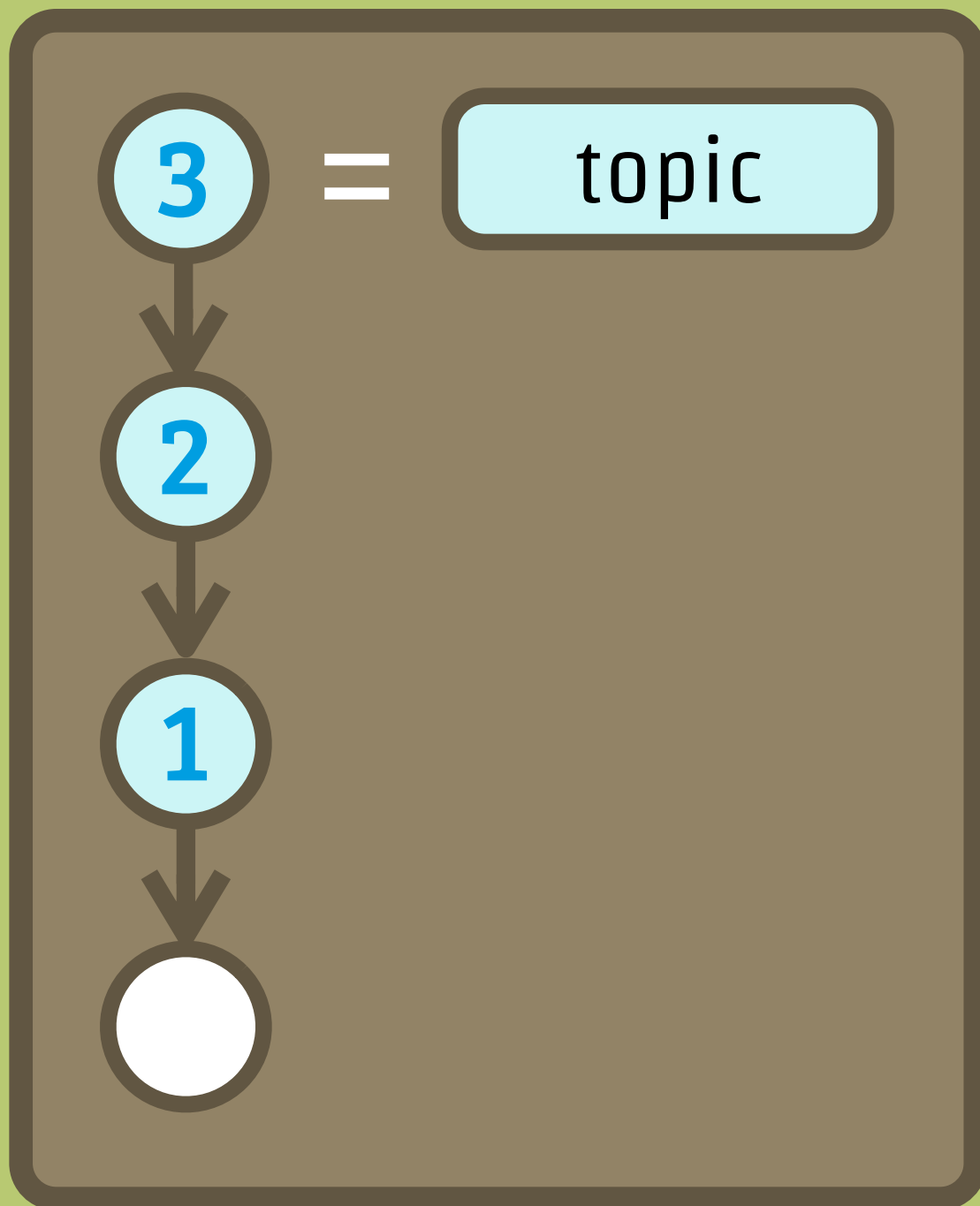
3

「さて、こいつの前のコミットは…」

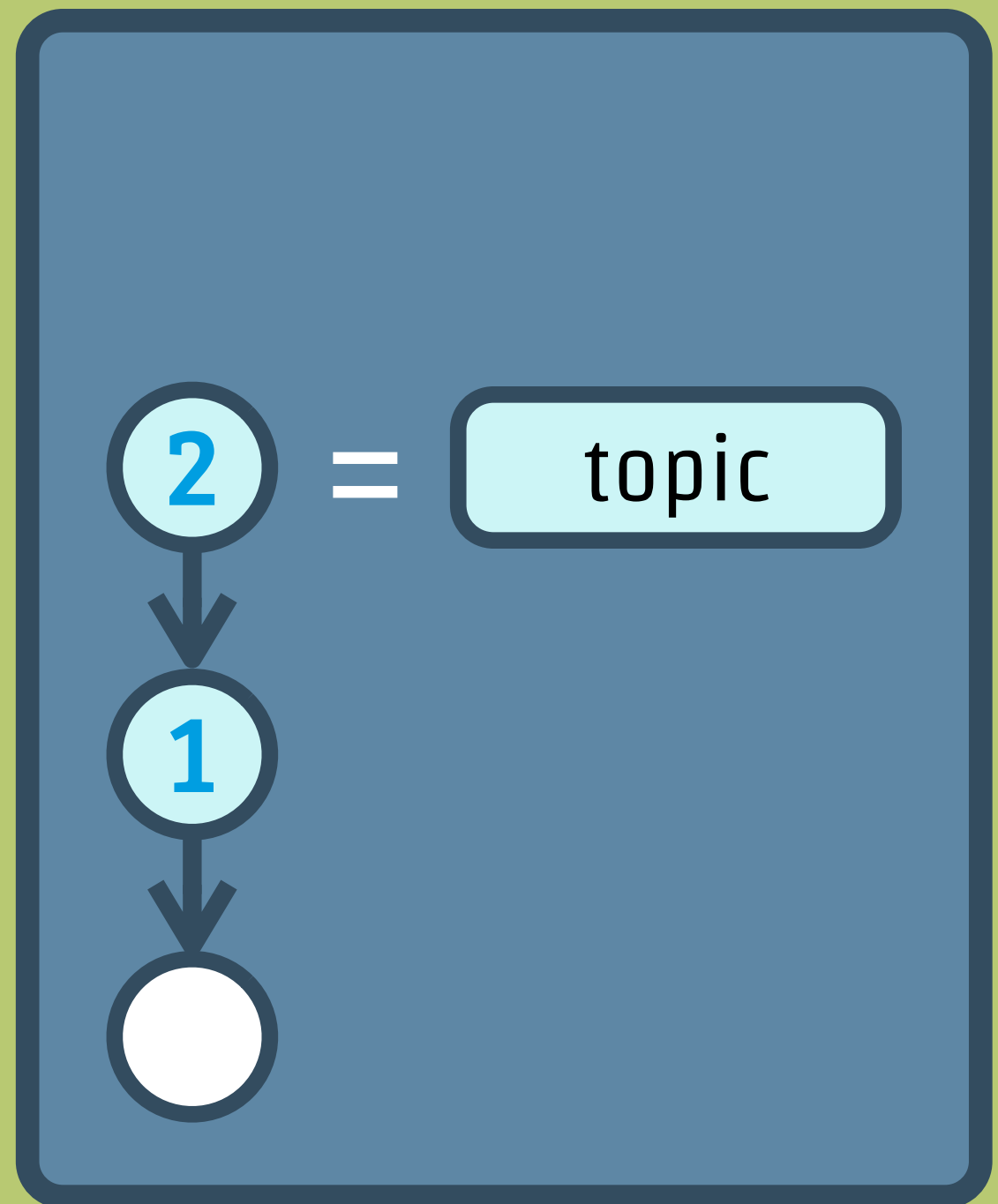


origin 「お、2番なら知ってるぞ！」

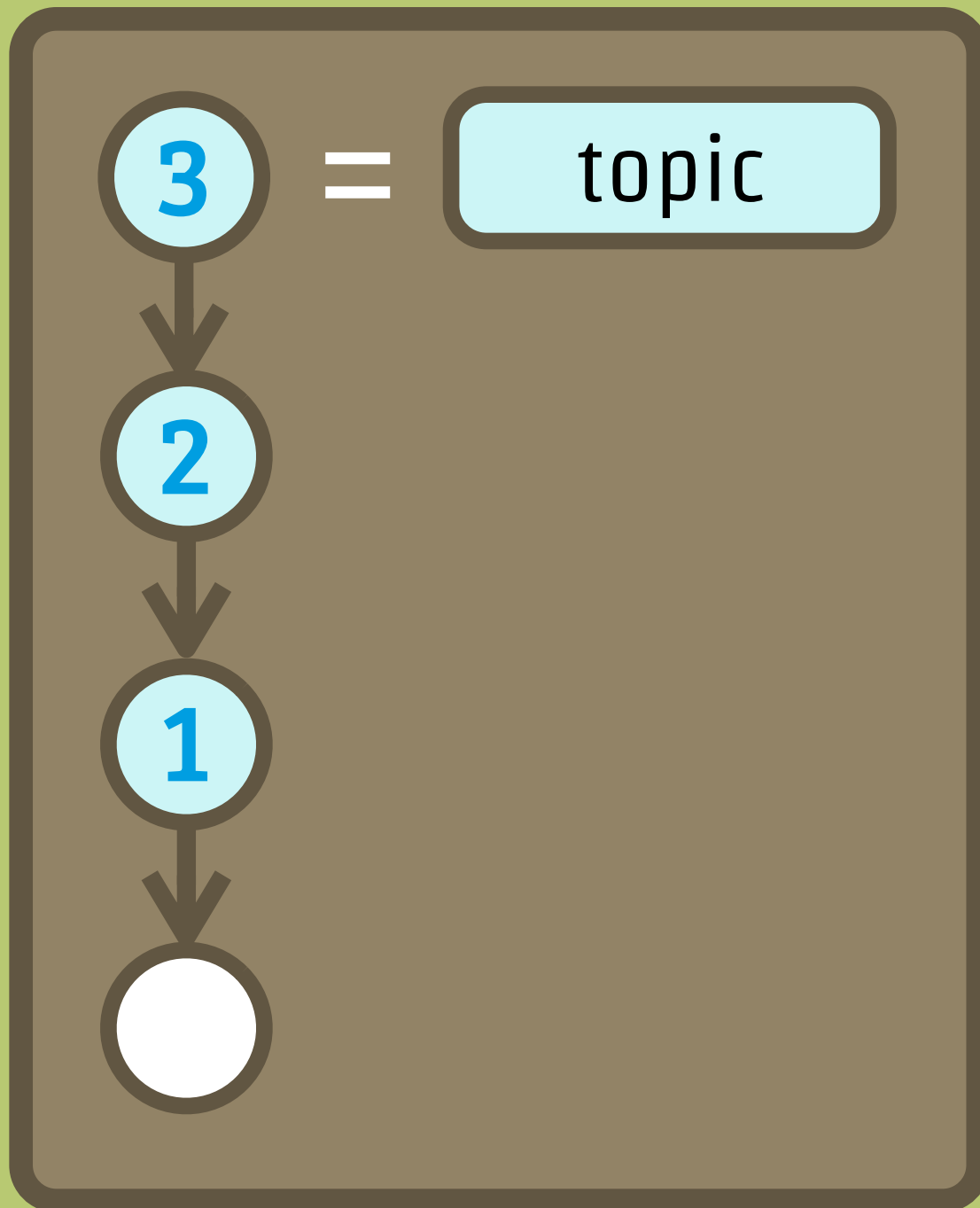
local



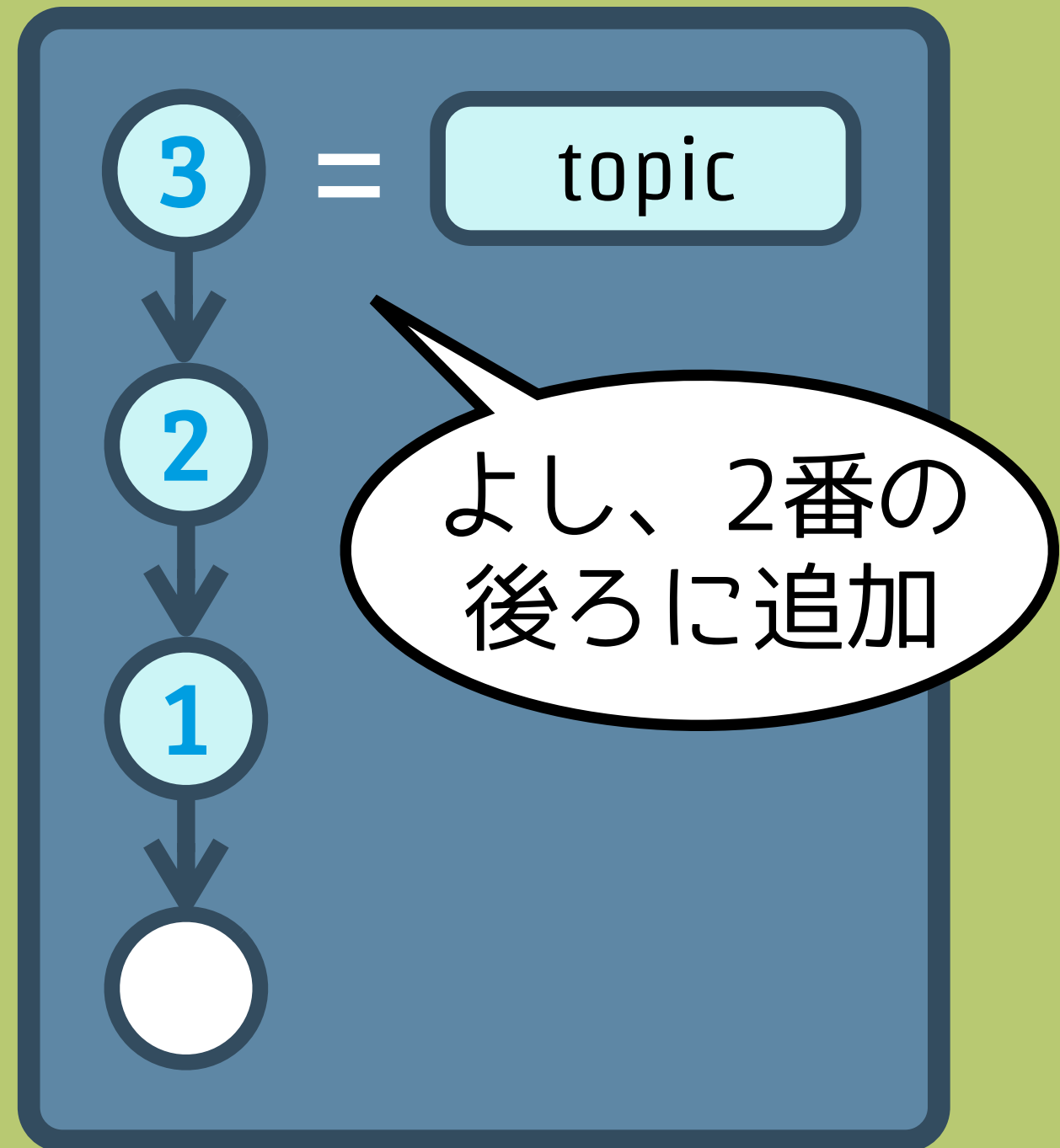
origin



local

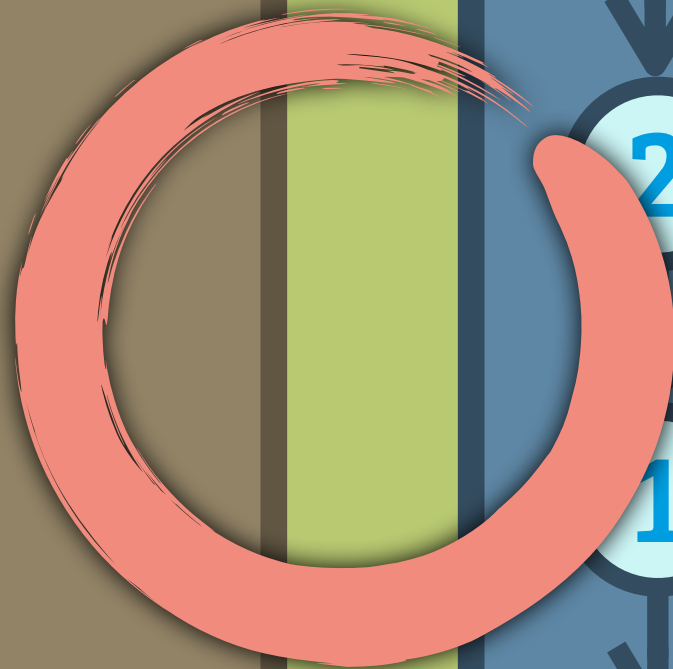
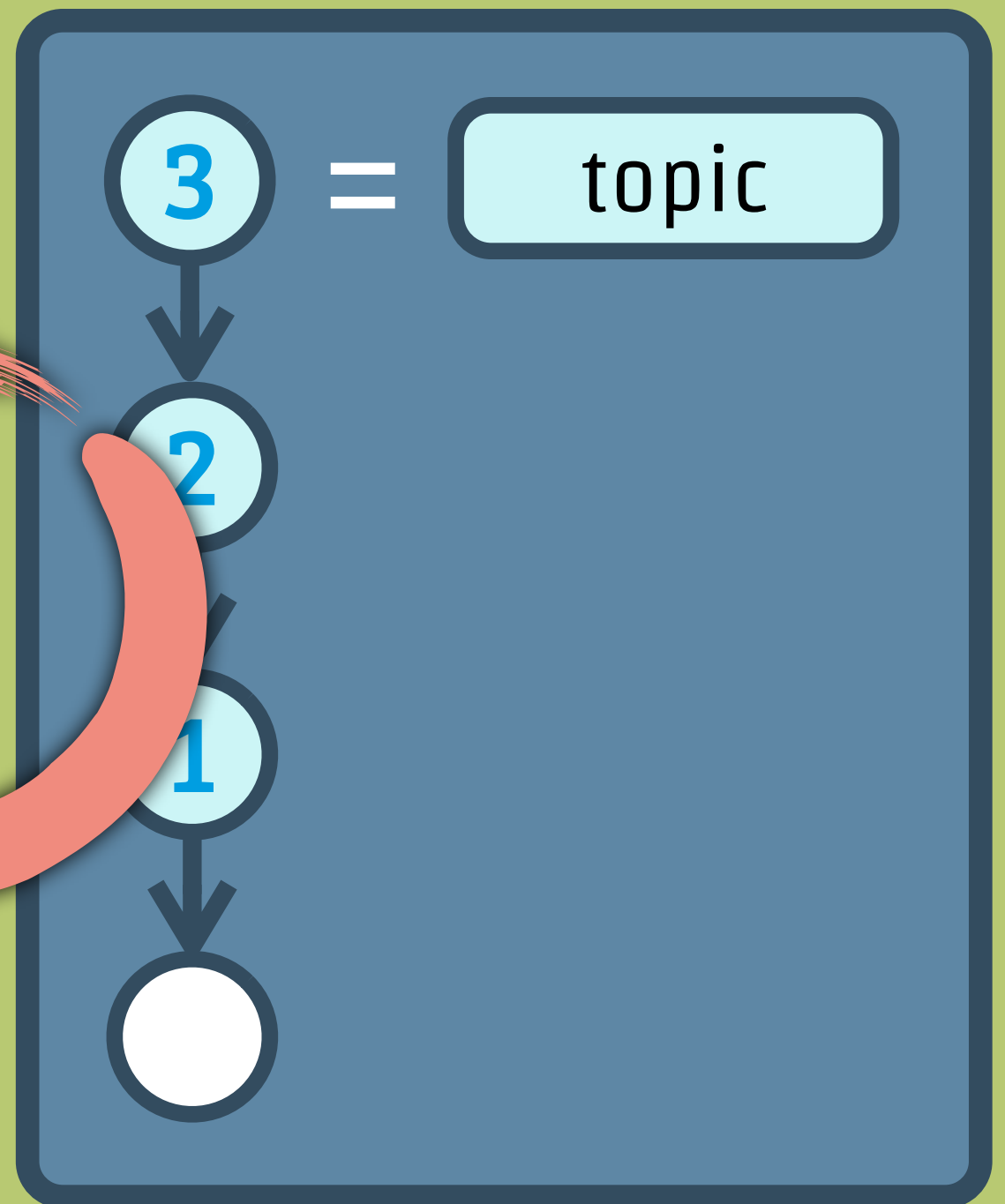
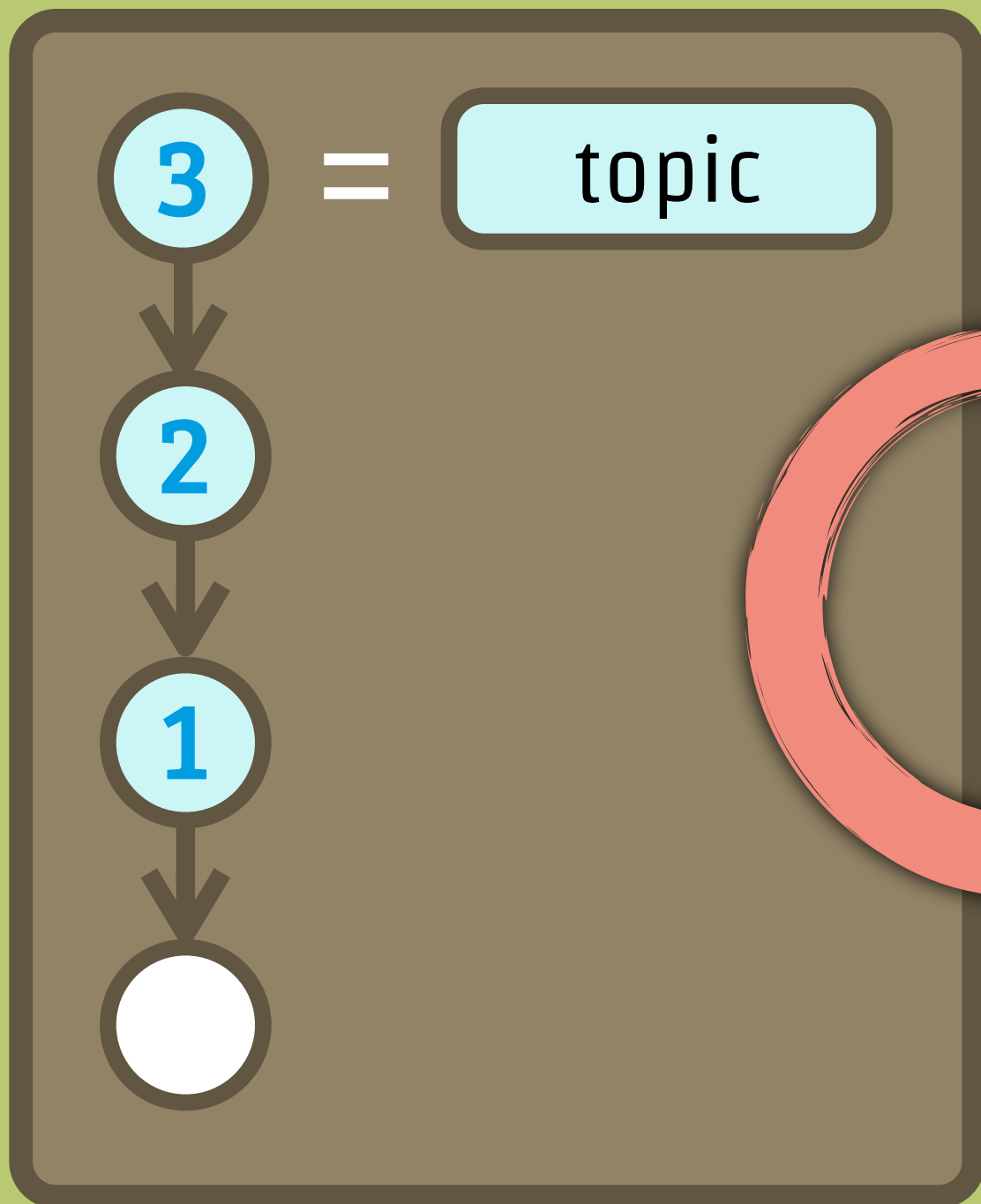


origin



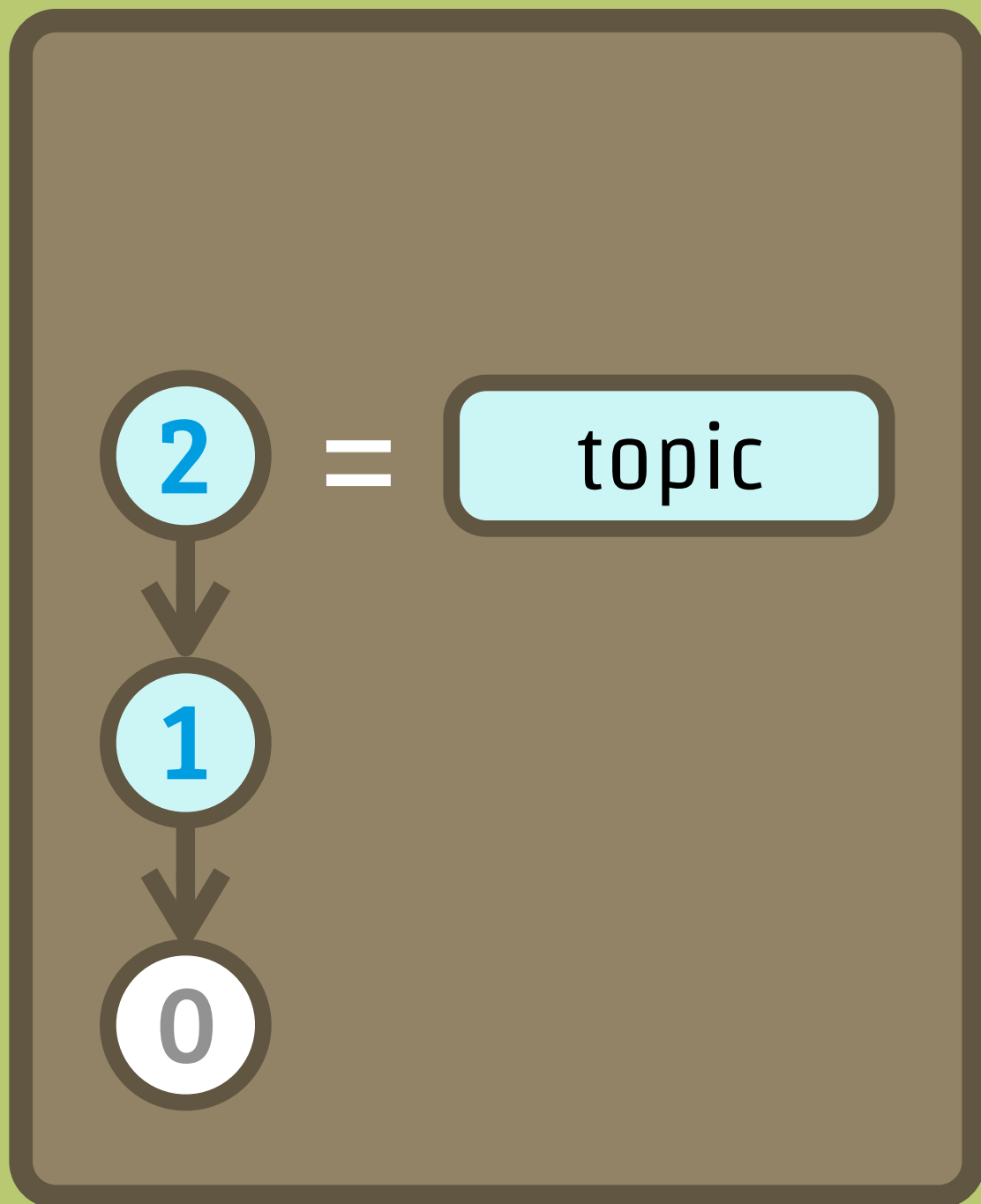
local

origin

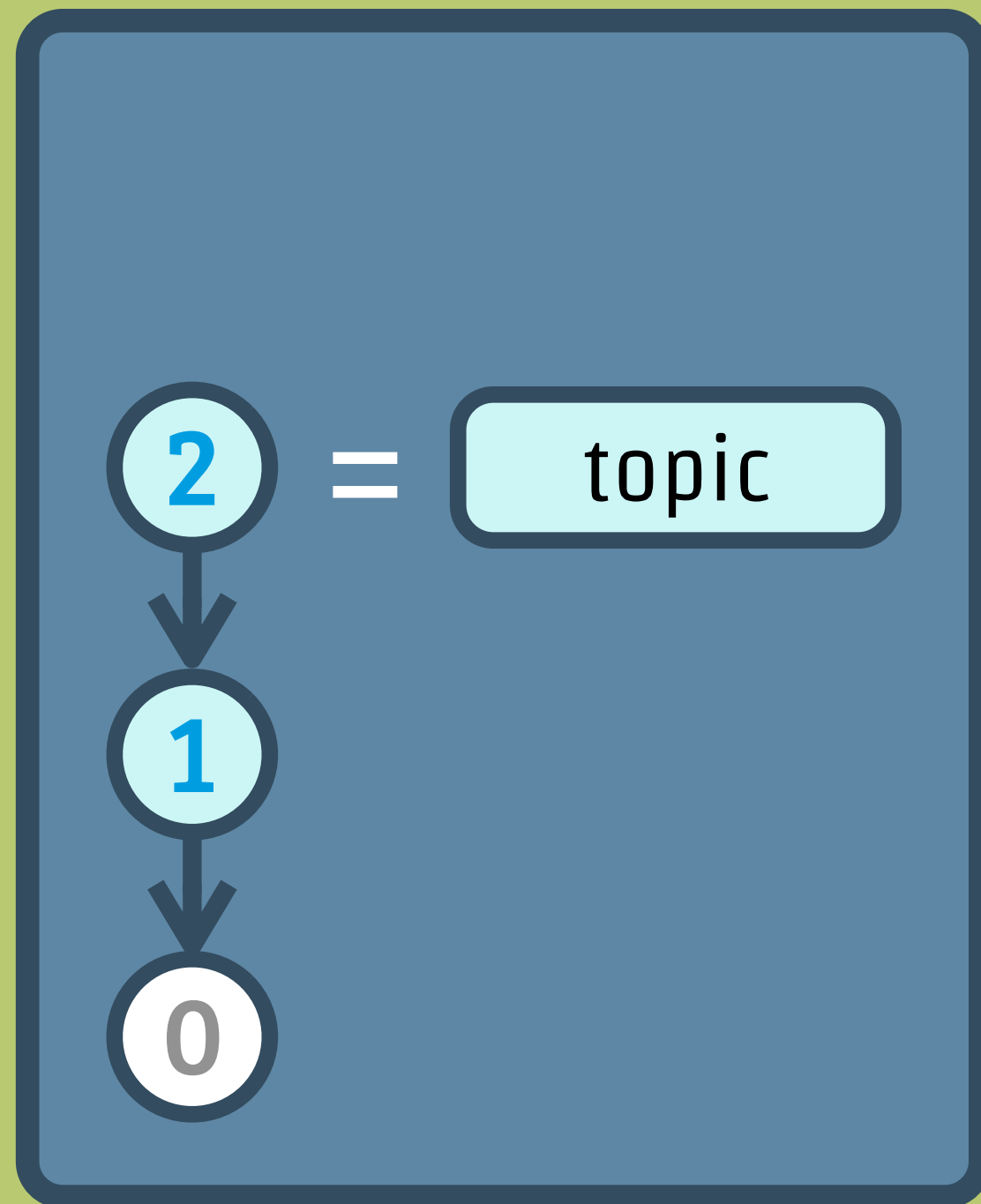


push after rebase

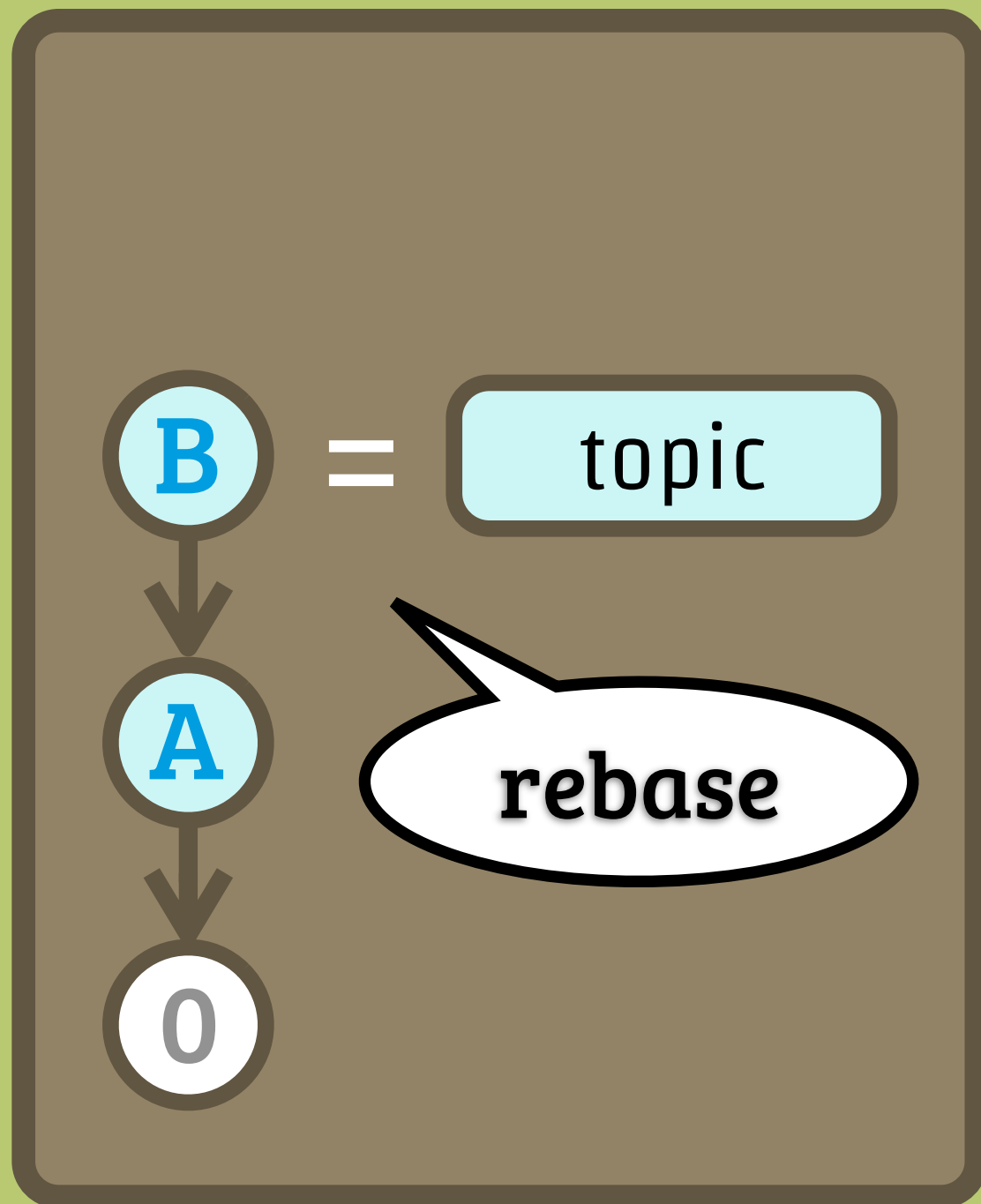
local



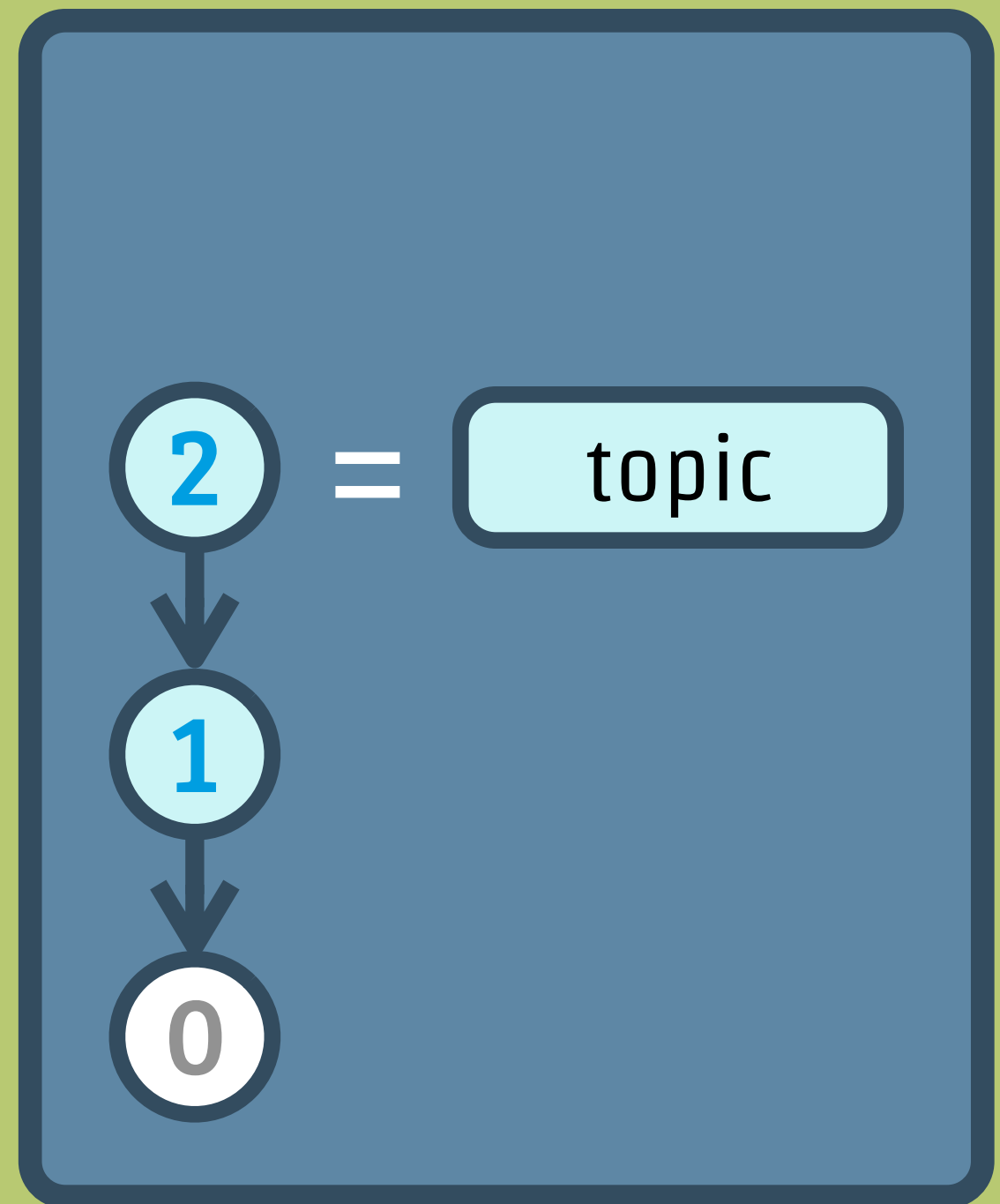
origin



local

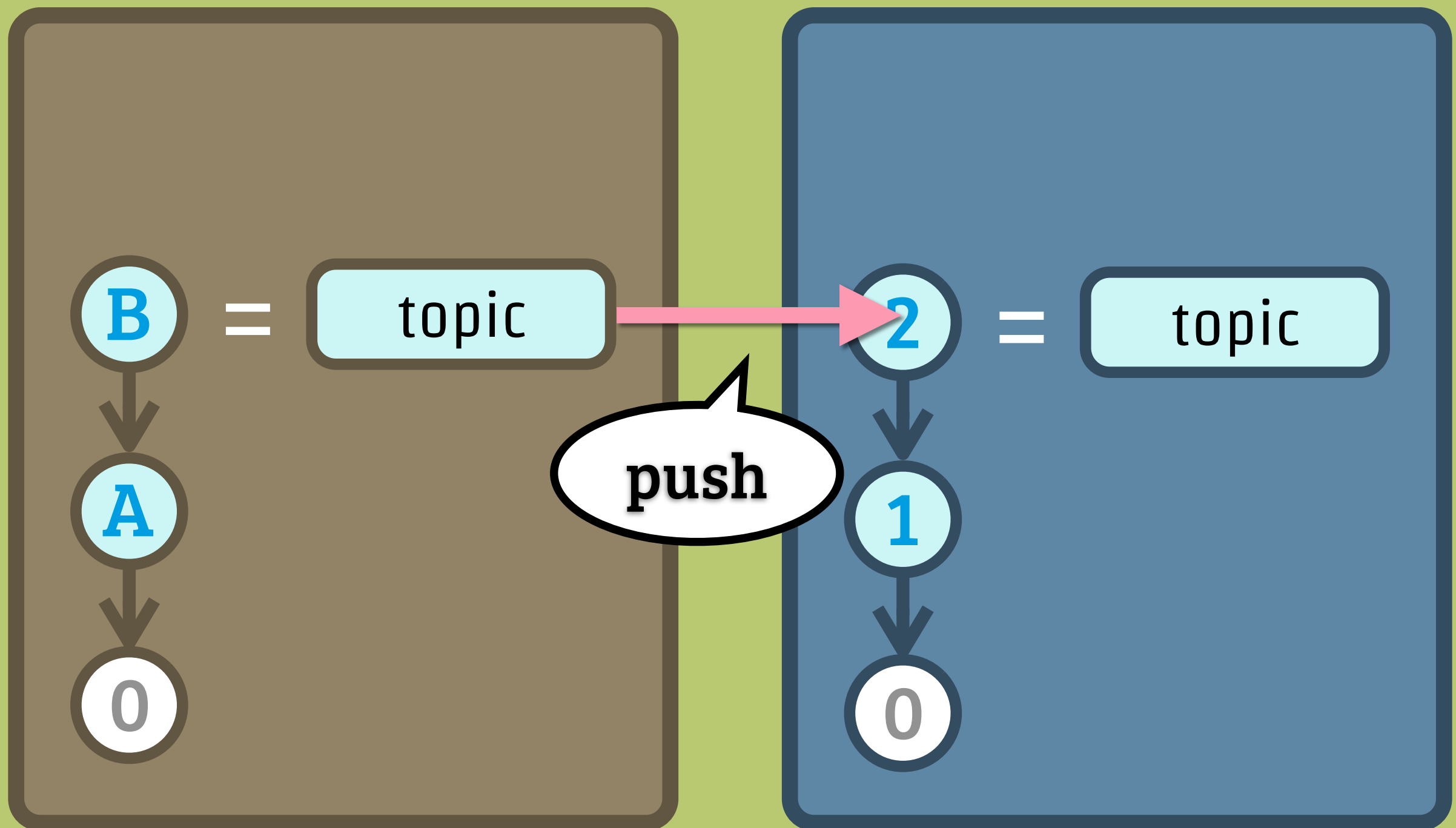


origin



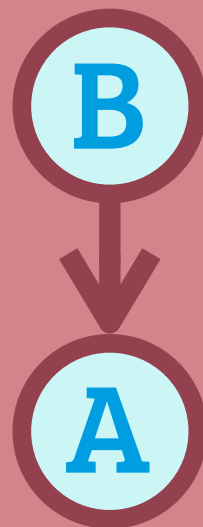
local

origin



origin 「お、pushがきたぞ？
どれどれ、内容を見てみよう」

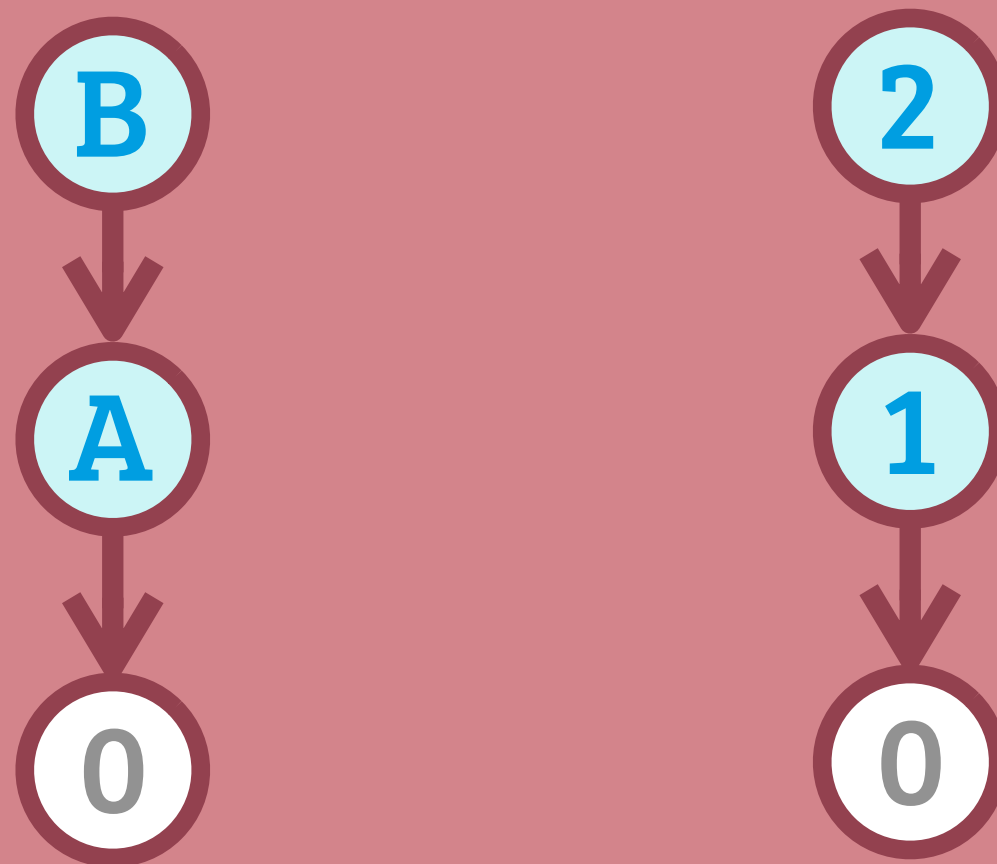
origin 「新しいコミットは2つか」



「さて、こいつの前のコミットは…」



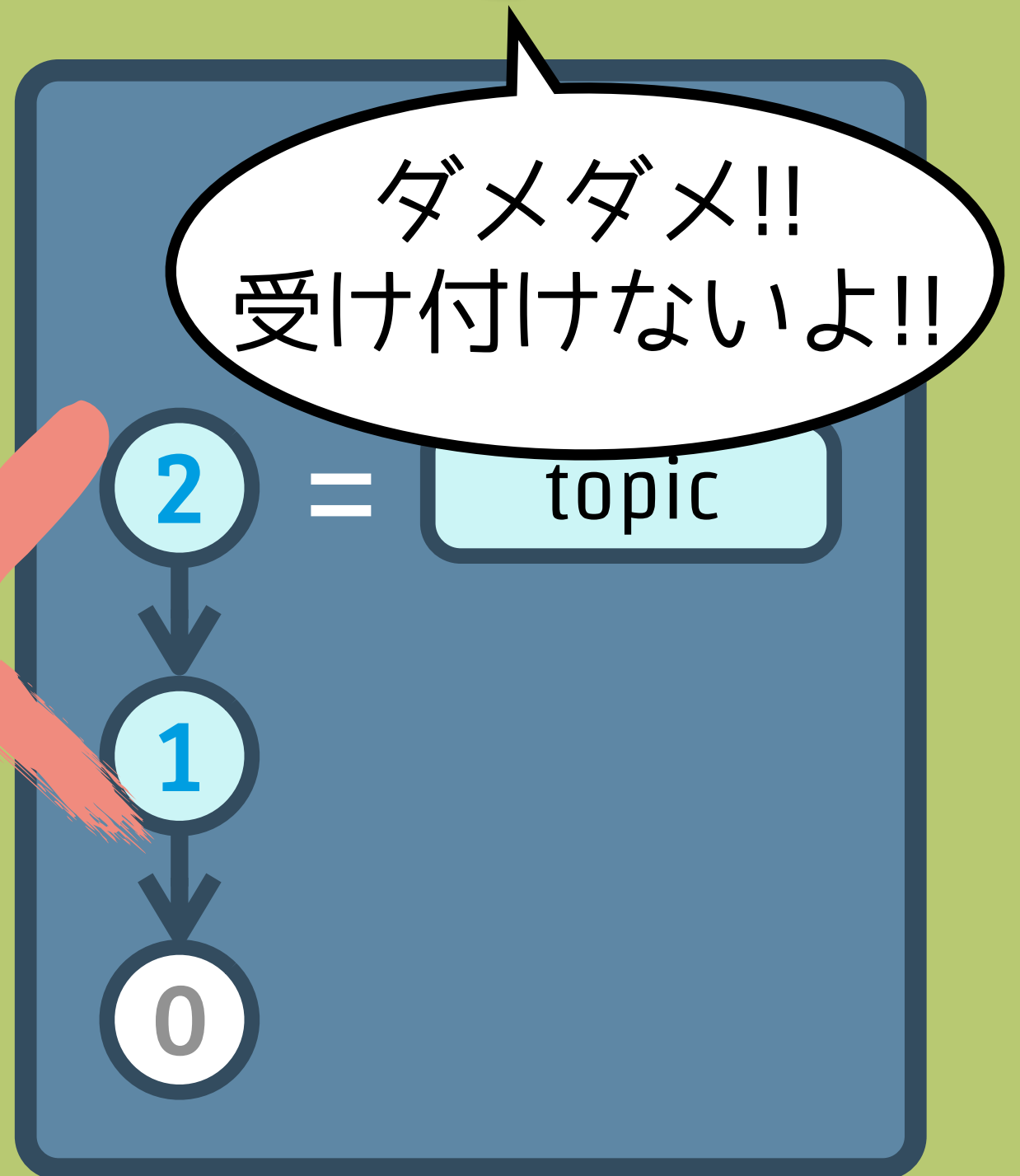
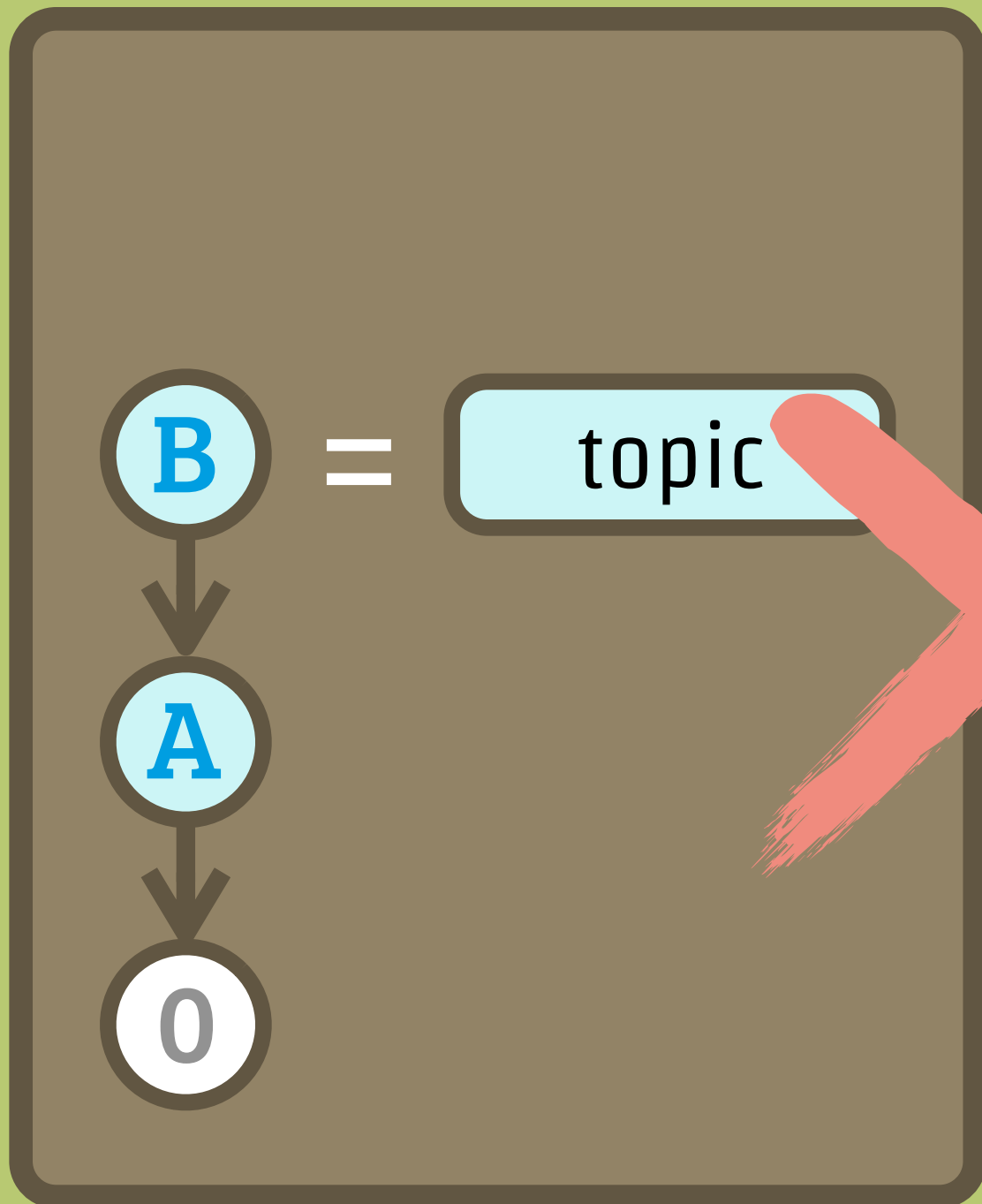
origin 「あれ？ 0番か…」



origin 「もう0番には次のコミットがあるし、1とAはリビジョンも違う!!」

local

origin



Point

push されているブランチを

rebase すると

push できなくなる

Point

よって、共有のブランチでは
rebase してはいけない !!

push -f ダメ! 絶対!!

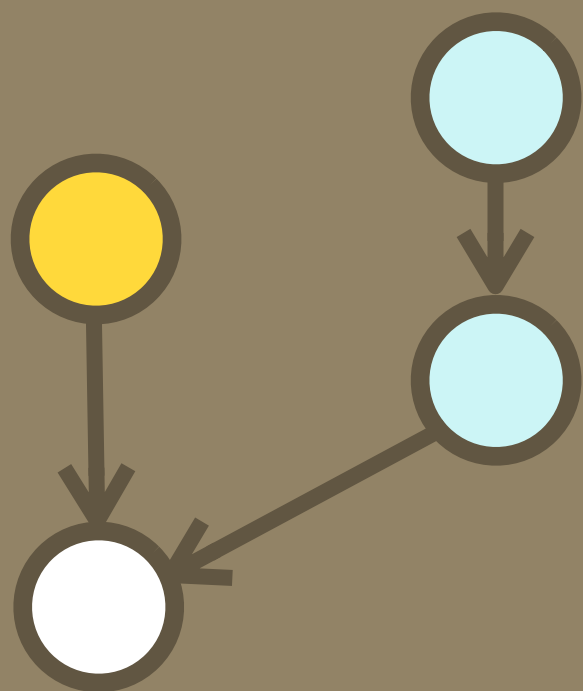
push -f で強制的に上書き push できますが
他の人が pull する時にマージする必要があったり
コミットログがおかしな事になるのでやめましょう

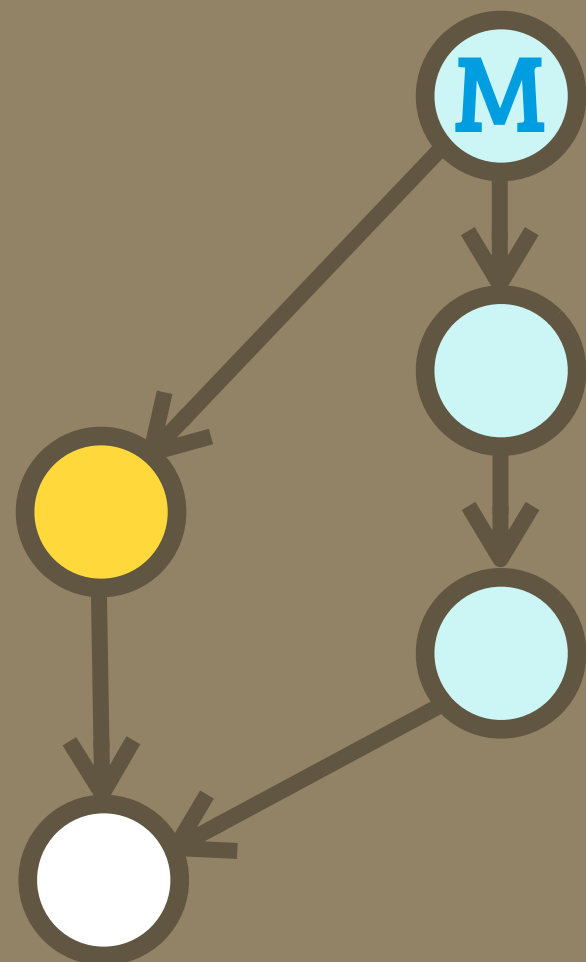
ブランチを master に追従するときに
使ってます!! マージはなんか怖いし...

＼怖くないよ／

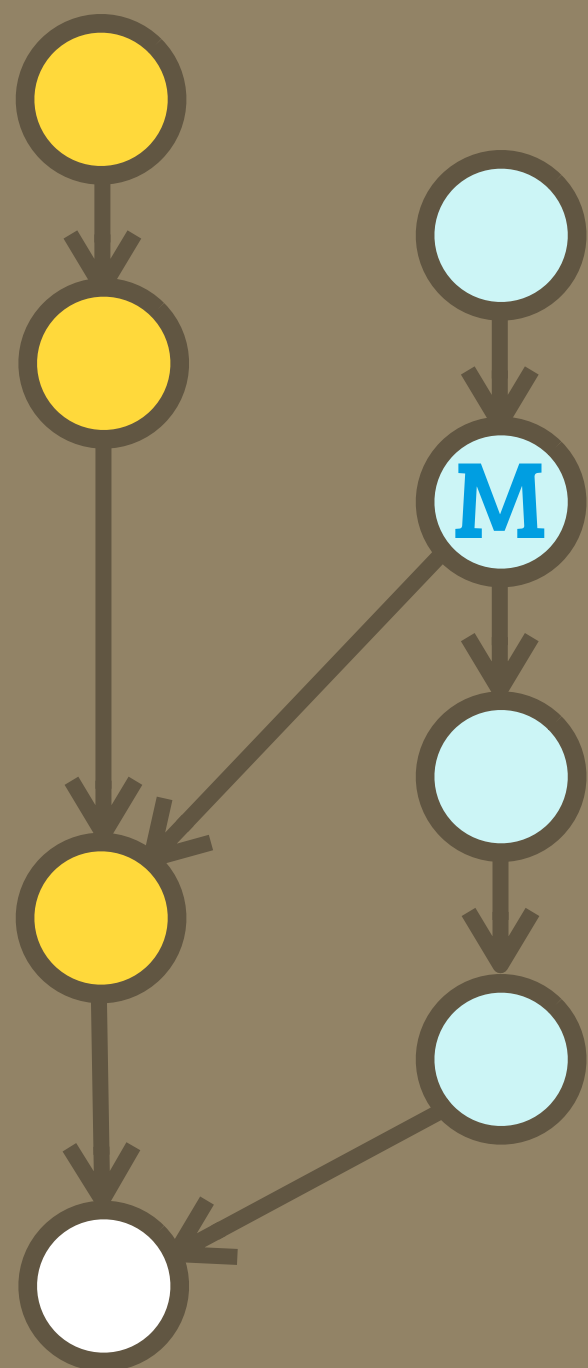


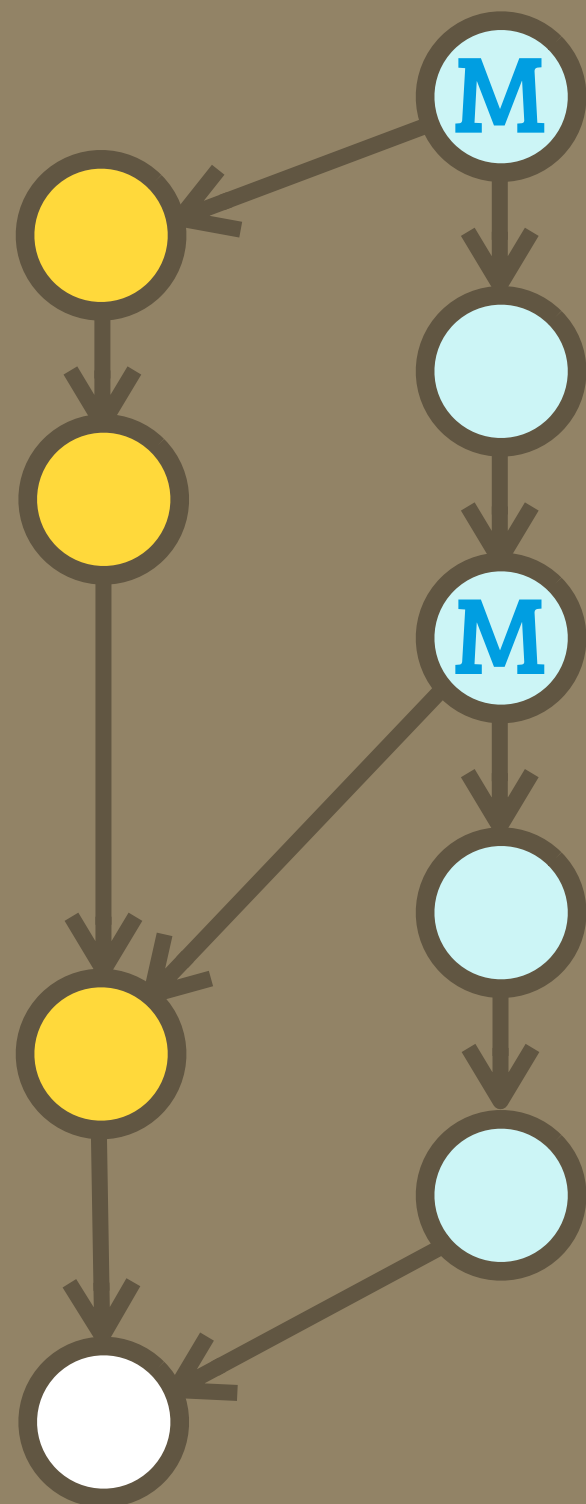
git



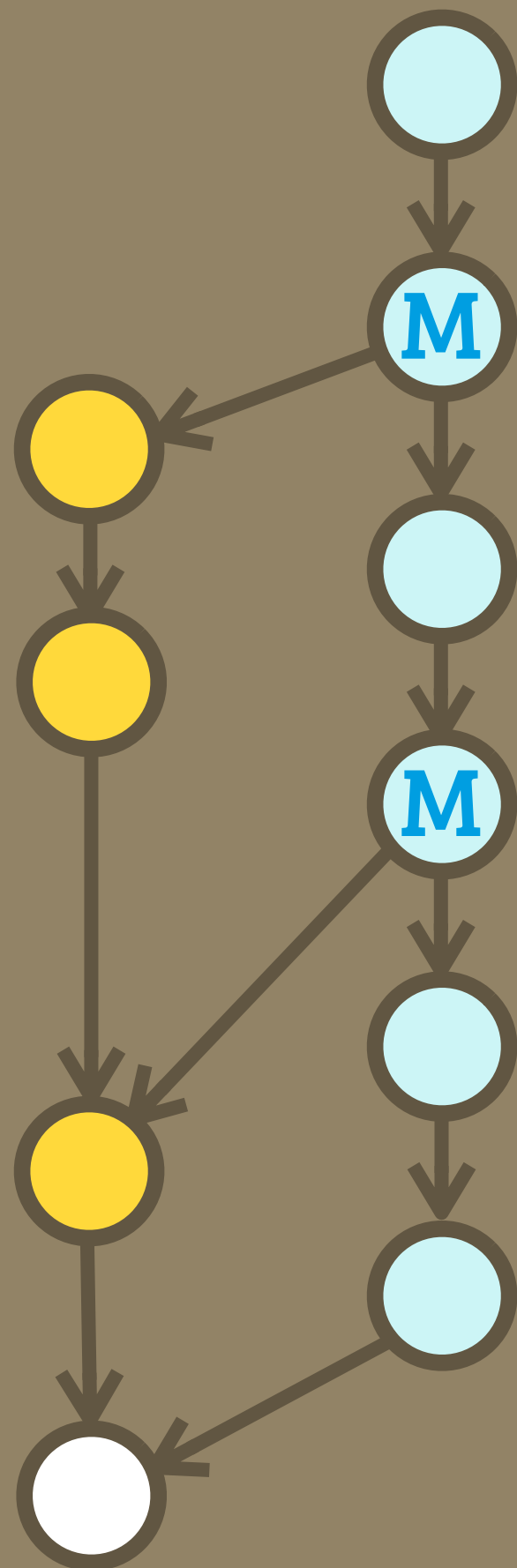


git merge **master**

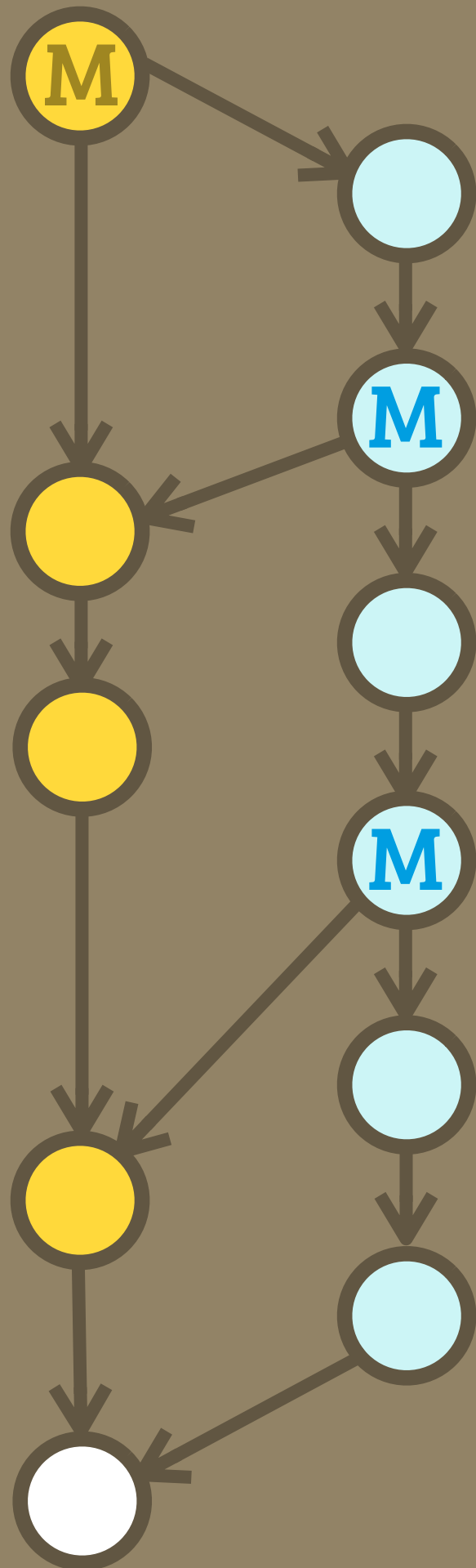


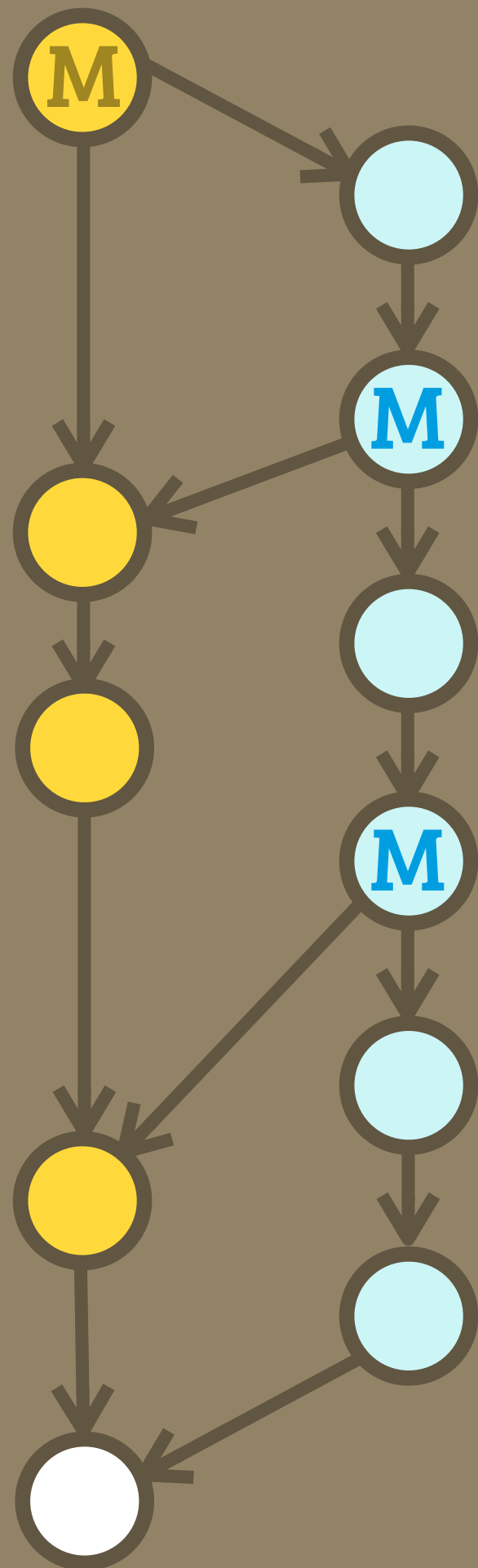


git merge **master**



git merge topic



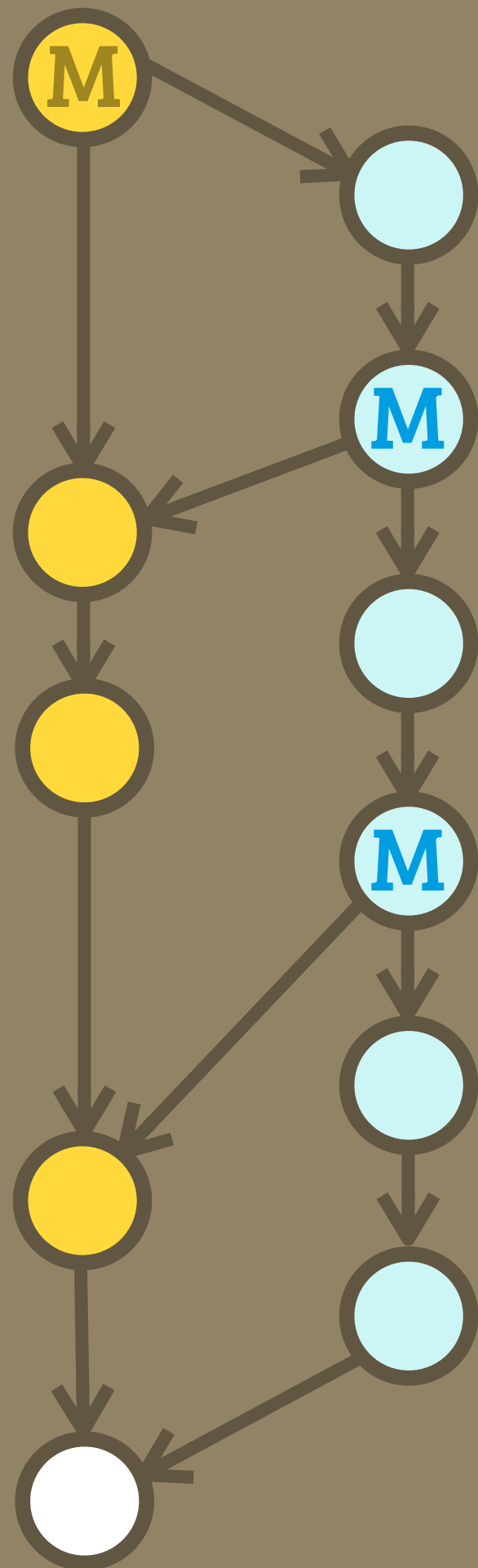


● と ● で衝突する修正を
しない限り、マージ時には
一切コンフリクトしません！

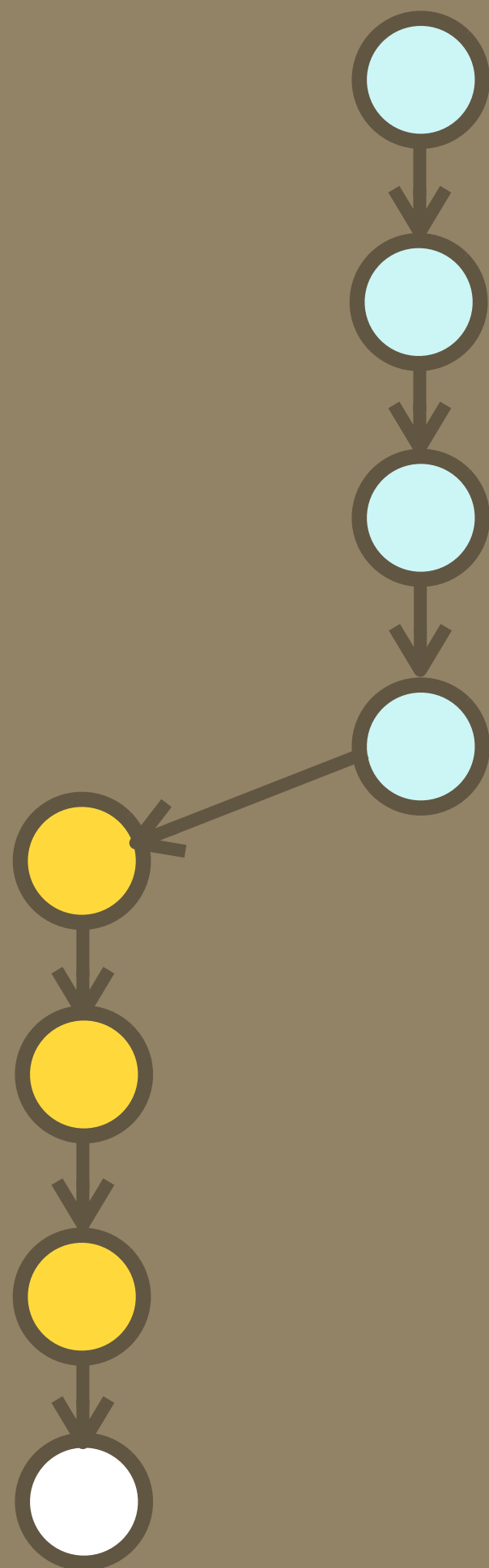
Point

git のマージは
相当かしこい

rebase すると、コミットログが
キレイになって見やすいよね!!

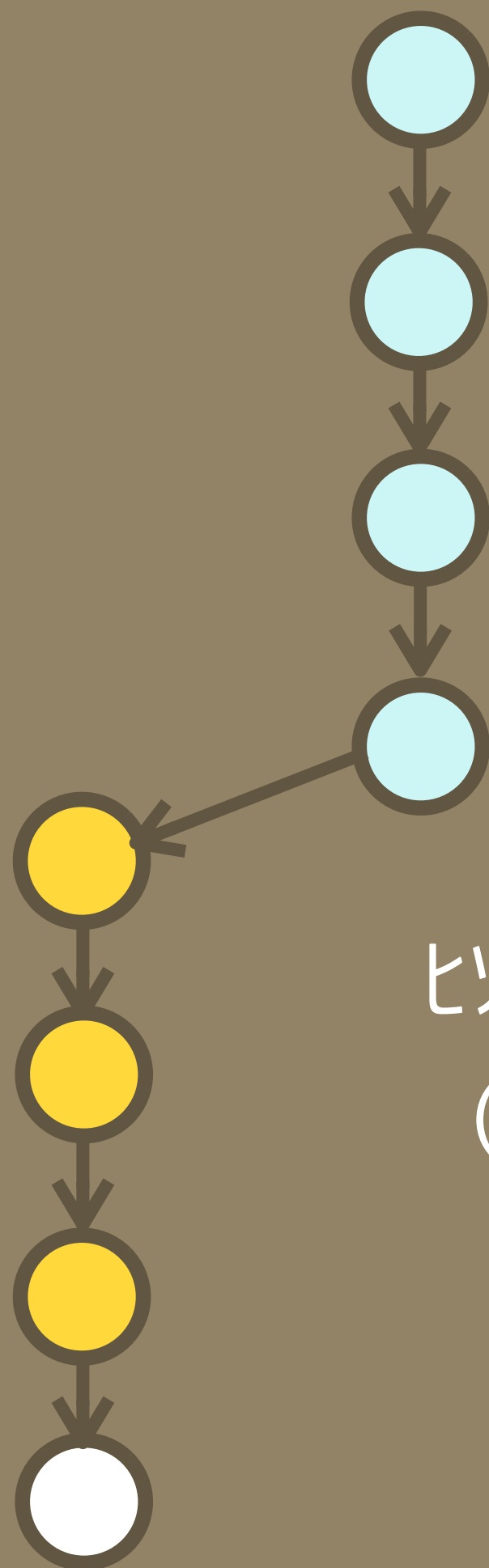


この一連の流れを
もし rebase でやっていると



確かに一直線でキレイ

お気づきだろうか...

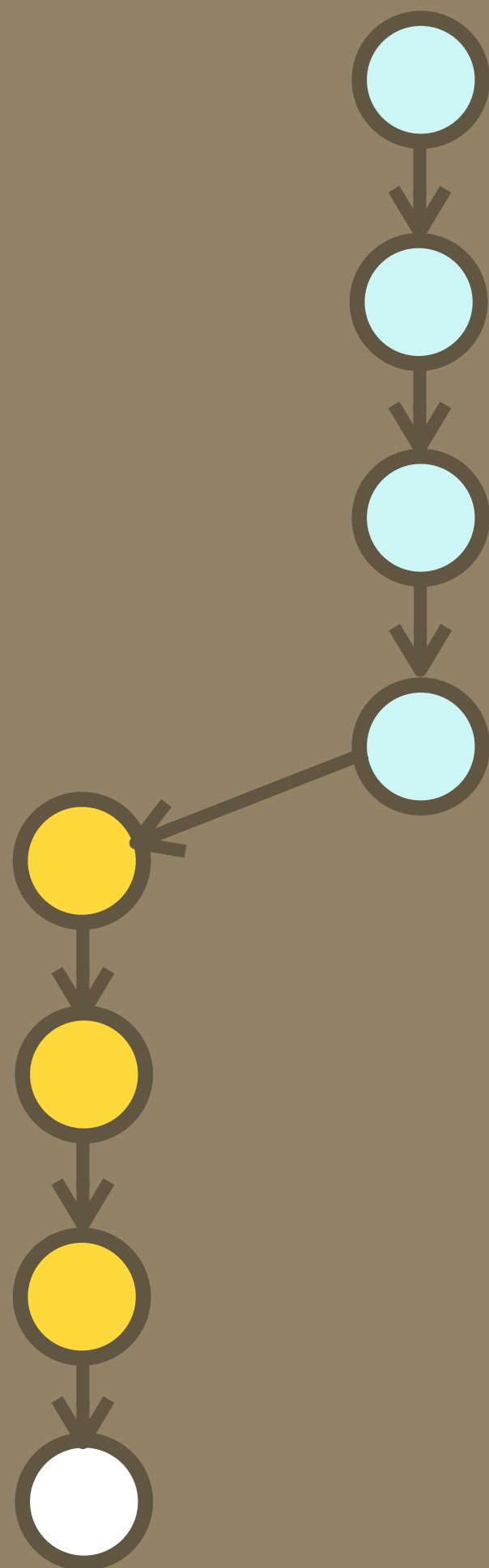


ヒソヒソ...

(なあ、もしかして
俺たち忘れられてないか?)



マージの記録は残らない



えっ



Point

rebase でコミットグラフは
一直線にキレイになるが
その陰で失われる情報がある

Point

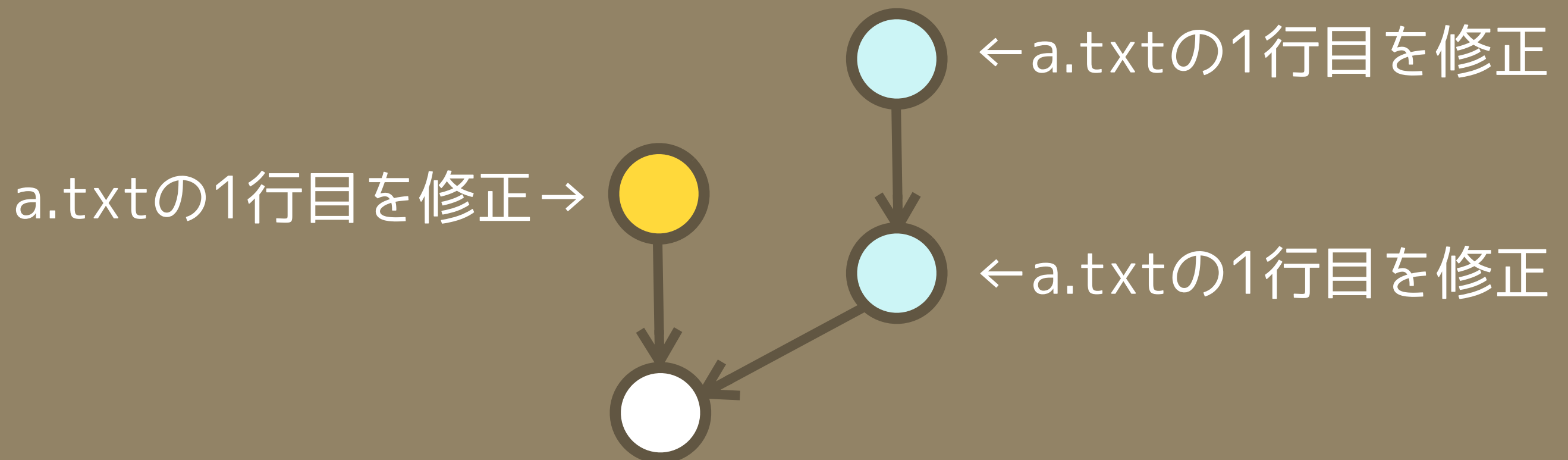
そもそも git で

入り組んだグラフになっても

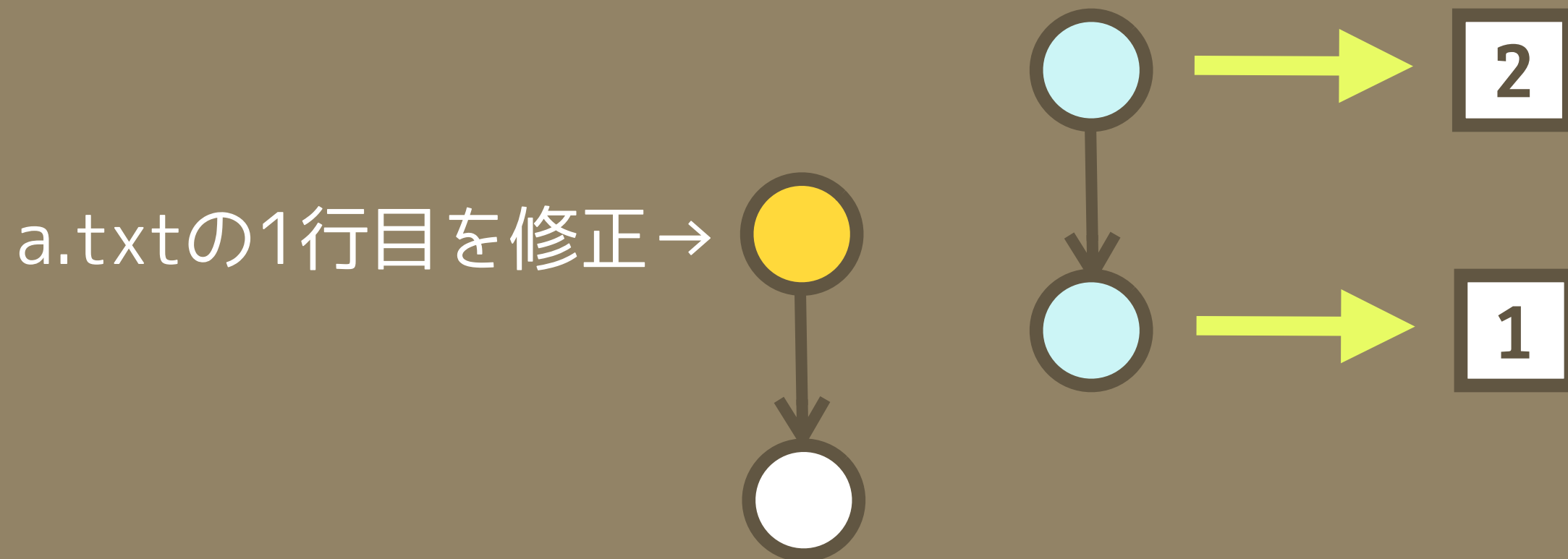
そんなに気にすることはない

merge よりも rebase の方が
マージコミットもなくて楽でしょ？

ここから `git rebase master` すると...

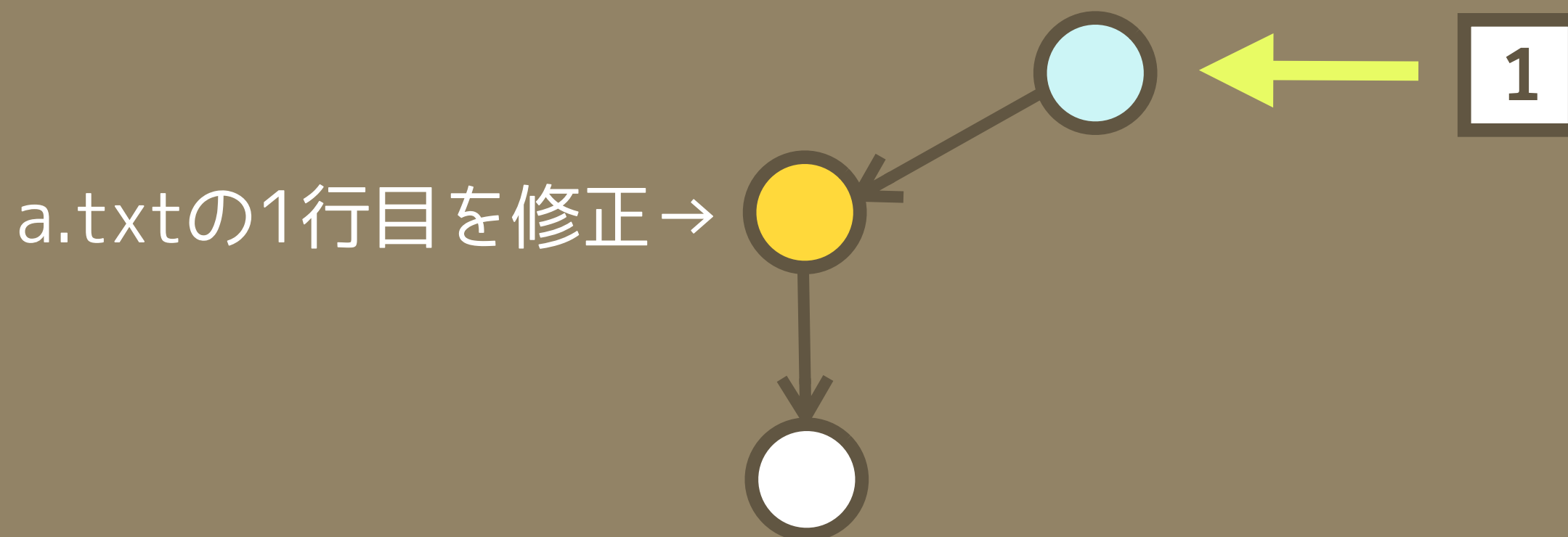


コミットがそれぞれパッチになって...



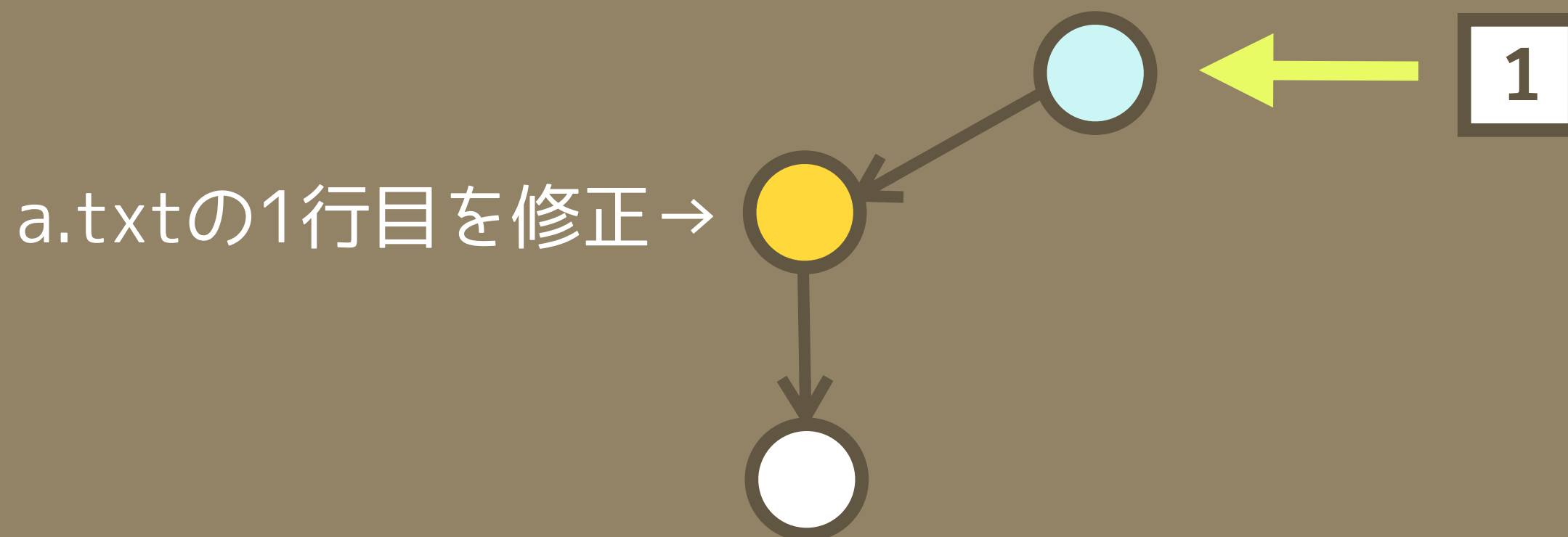
1つ目のパッチを適用

a.txtの1行目を修正するパッチ1つ目↓

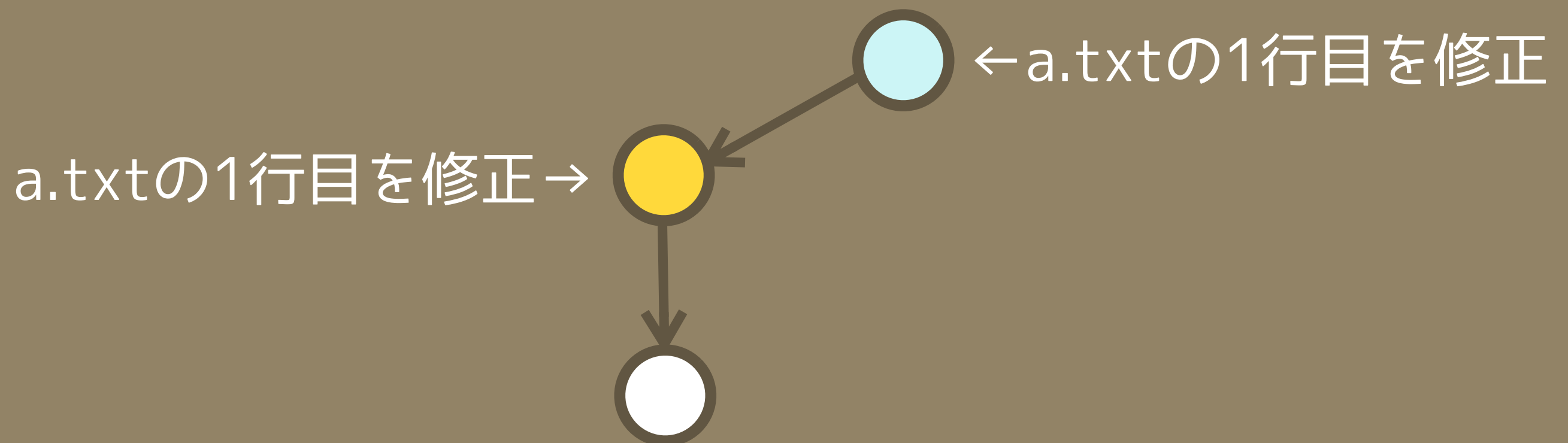


コンフリクト！

a.txtの1行目を修正するパッチ1つ目↓

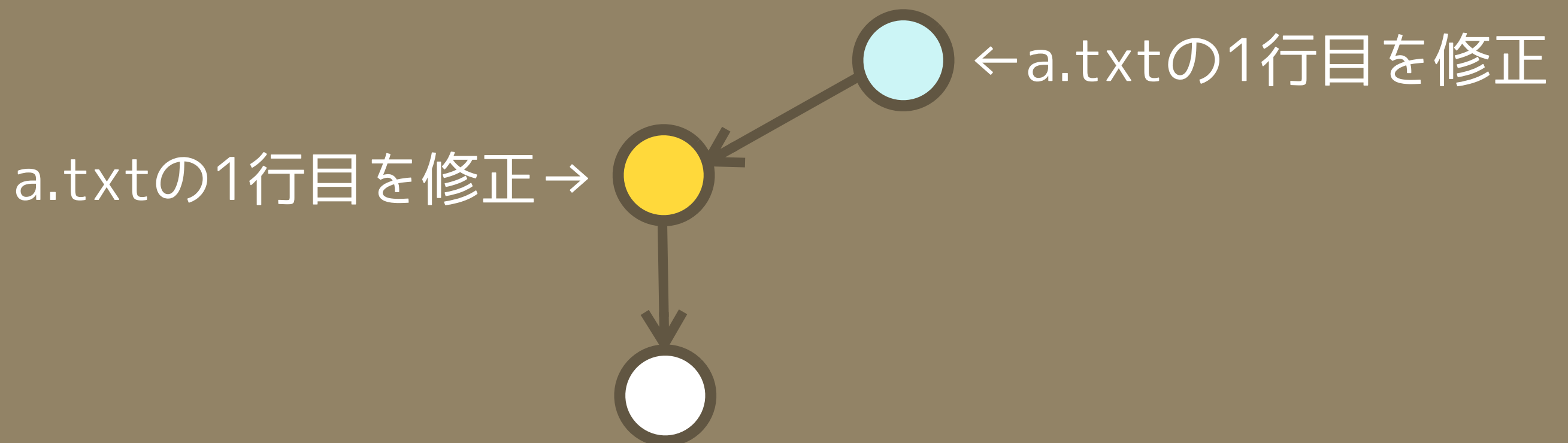
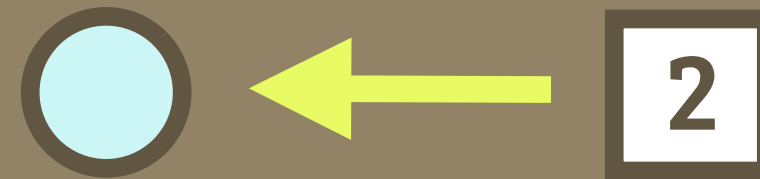


コンフリクトを解消



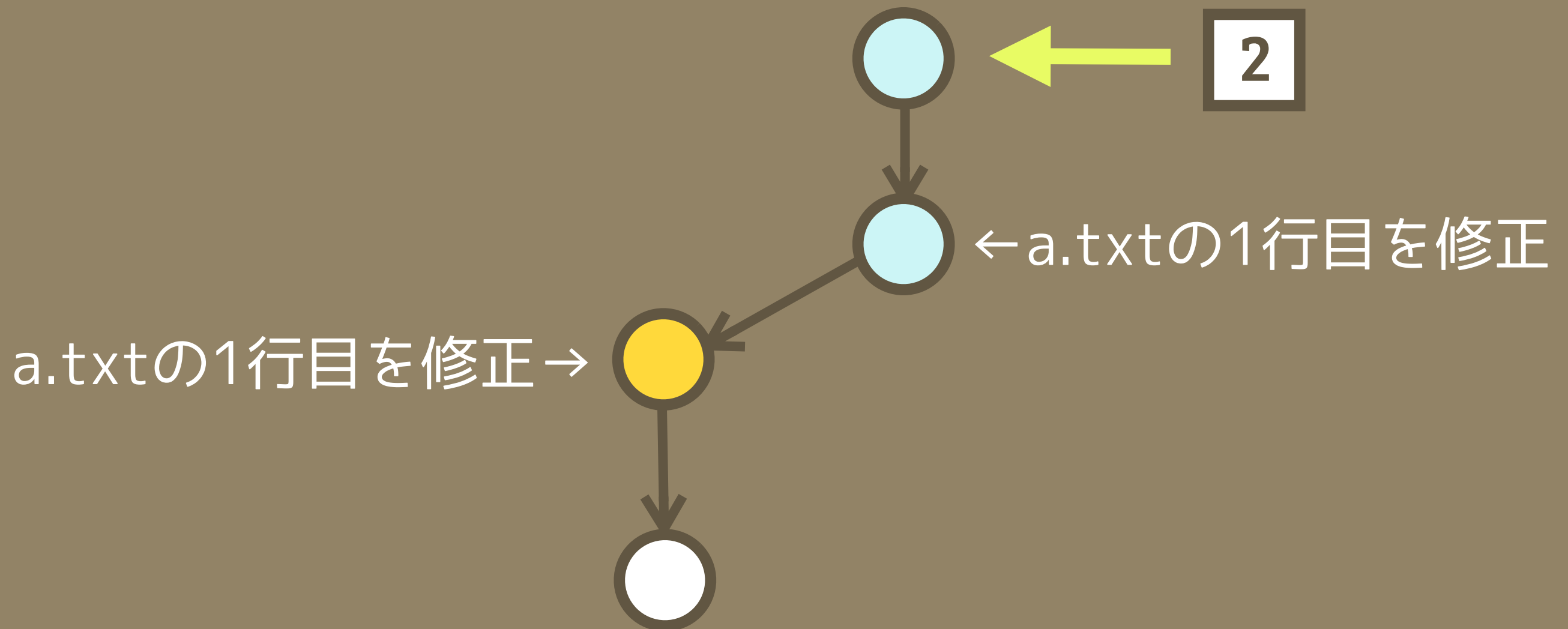
2つ目のパッチを適用

a.txtの1行目を修正するパッチ2つ目↓

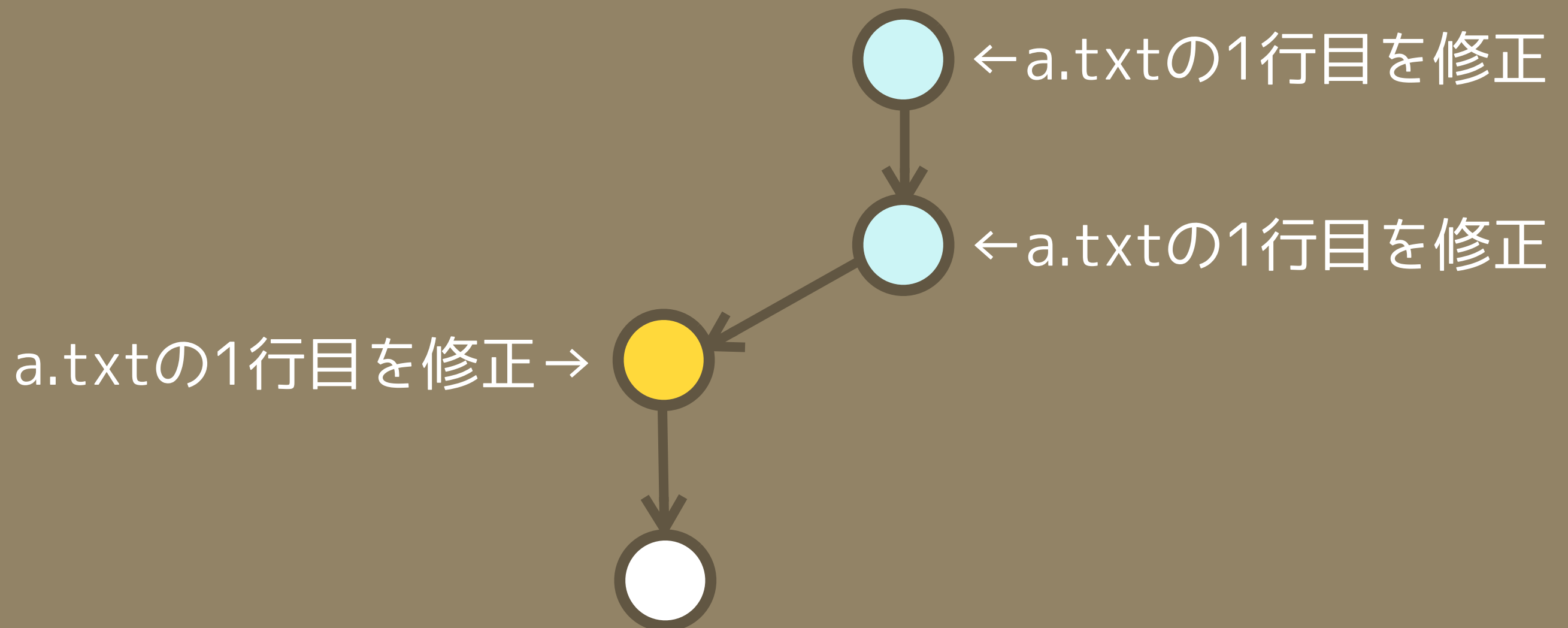


コンフリクト！

a.txtの1行目を修正するパッチ2つ目↓



コンフリクトを解消



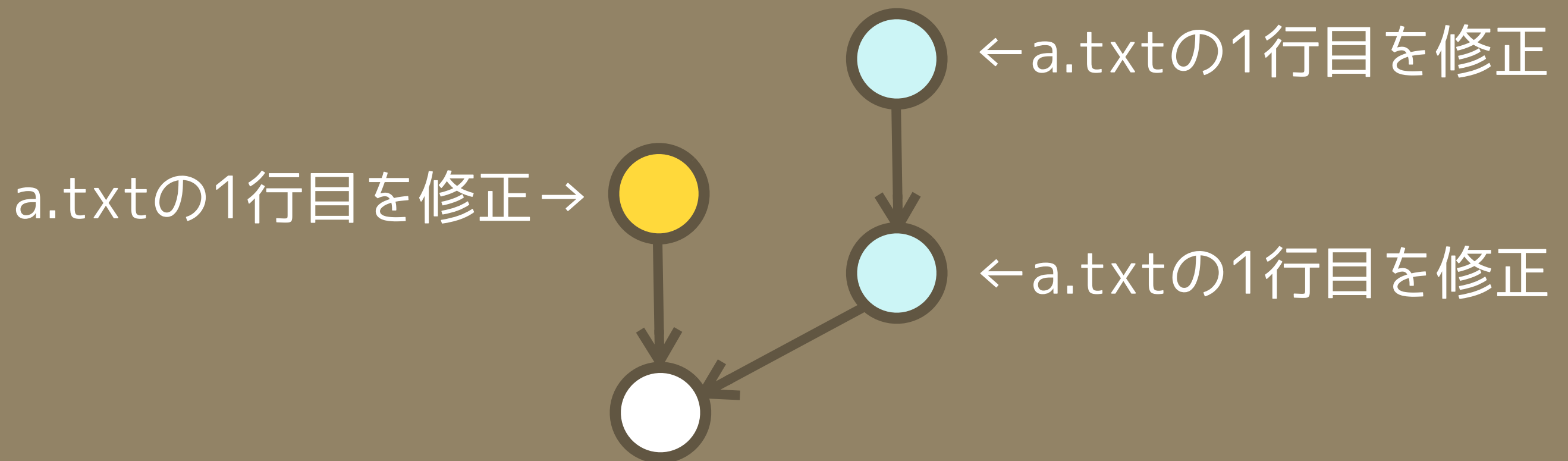
Point

rebase だと

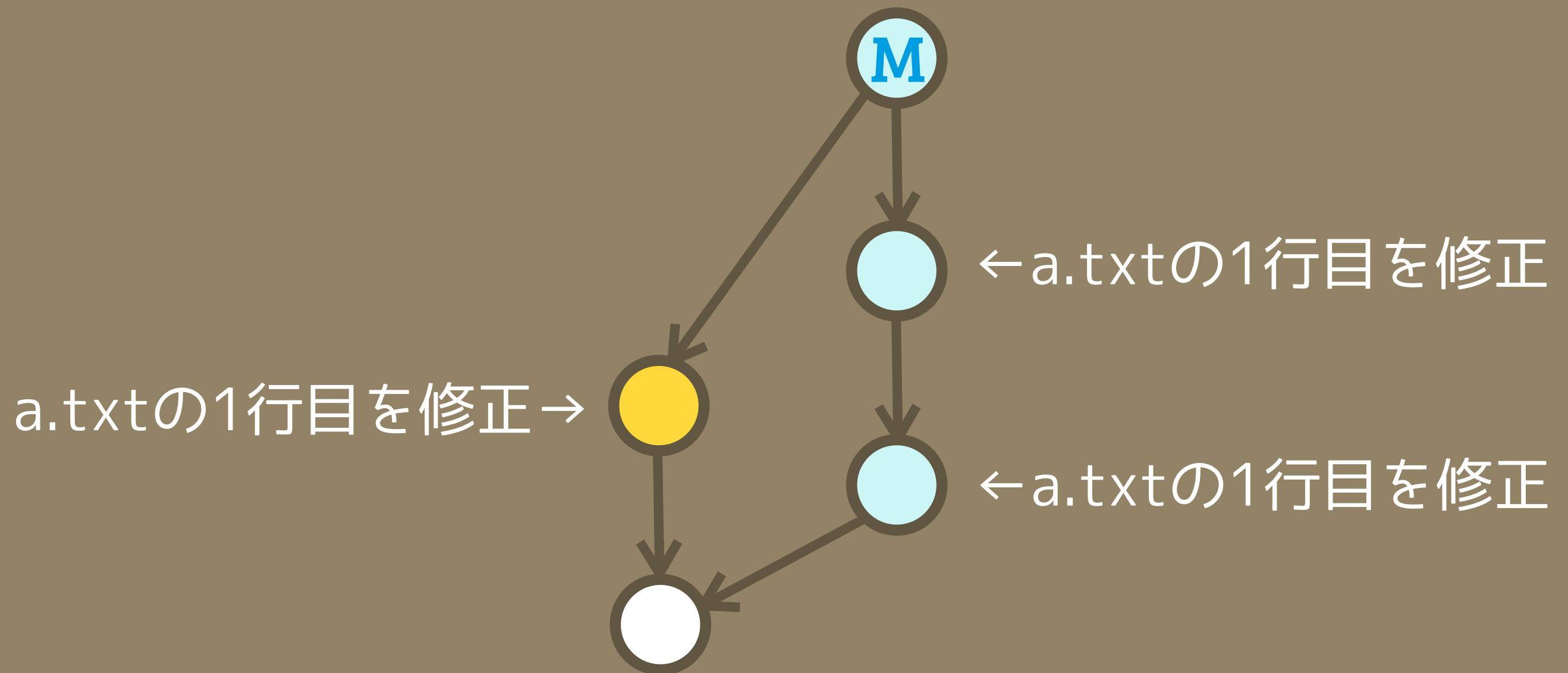
コミットそれぞれについて

コンフリクトの解消が必要

ちなみに `git merge master` だと...



コンフリクトを解消



Point

merge だと

コミットがいくつあろうと

コンフリクト解消は1回だけ

Point

したがって

rebase よりもむしろ

merge の方が楽です

リベースの功罪

Good

- ・ コミットグラフがキレイになる
 - マージ後のログがキレイになるので
master にマージする直前にやるのは
OK とする事がある
 - GitHub などのオープンソースプロジェクトに
プルリクエストを送る場合は
rebase してから送るのがマナーとされている

リベースの功罪

Bad

- ・ push されたブランチをリベースすると push できなくなる
- ・ 「マージした」という事実は失われる
- ・ マージに比べるとコンフリクト解消が面倒

すこしは git の
理解の手助けに
なれたでしょうか？

＼怖くないよ／



git

＼ズツ友だよ………!!／



git

Have a Nice Git!

References

Pro Git book

<http://git-scm.com/book/ja/>

git merge or rebase, ff or no-ff - Togetter

<http://togetter.com/li/407277>

A successful Git branching model

<http://nvie.com/posts/a-successful-git-branching-model/>